



Algoritmos de Ordenação



Prof. Luiz Gustavo Almeida Martins

Introdução

- ▶ **Ordenação** é a tarefa de colocar um conjunto de dados em uma **determinada ordem**
 - ▶ **Outras denominações:** classificação e organização
 - ▶ Permite o **acesso mais eficiente** aos dados

Introdução

- ▶ **Ordenação** é a tarefa de colocar um conjunto de dados em uma **determinada ordem**
 - ▶ **Outras denominações:** classificação e organização
 - ▶ Permite o **acesso mais eficiente** aos dados
- ▶ É importante para **melhorar a eficiência** de outros processos computacionais
 - ▶ **Ex:** métodos de busca e intercalação (fusão), banco de dados, etc.

Introdução

- ▶ **Ordenação** é a tarefa de colocar um conjunto de dados em uma **determinada ordem**
 - ▶ **Outras denominações:** classificação e organização
 - ▶ Permite o **acesso mais eficiente** aos dados
- ▶ É importante para **melhorar a eficiência** de outros processos computacionais
 - ▶ **Ex:** métodos de busca e intercalação (fusão), banco de dados, etc.
- ▶ Os algoritmos de ordenação trabalham sobre **registros**
 - ▶ O **campo chave** é usado para controlar a ordenação
 - ▶ Muito usado na apresentação de listagens

Exemplo

Arquivo original

Nome	Idade
João	15
Daniel	60
Maria	32

chave = nome



Arquivo ordenado

Nome	Idade
Daniel	60
João	15
Maria	32

Exemplo

Arquivo original

Nome	Idade
João	15
Daniel	60
Maria	32



chave = **idade**

Arquivo ordenado

Nome	Idade
João	15
Maria	32
Daniel	60

Introdução

- ▶ A **saída de um algoritmo de ordenação** deve satisfazer duas condições:
 - ▶ Ser uma **permutação da entrada**

Introdução

- ▶ A **saída de um algoritmo de ordenação** deve satisfazer duas condições:
 - ▶ Ser uma **permutação da entrada**
 - ▶ Estar em uma **ordem crescente ou decrescente**

Introdução

- ▶ A **saída de um algoritmo de ordenação** deve satisfazer duas condições:
 - ▶ Ser uma **permutação da entrada**
 - ▶ Estar em uma **ordem crescente ou decrescente**
- ▶ **Exemplos:**
 - ▶ **Entrada:** 6 5 7 1 2 4 3
 - ▶ **Entrada:** V U X Z Y

Introdução

- ▶ A **saída de um algoritmo de ordenação** deve satisfazer duas condições:

- ▶ Ser uma **permutação da entrada**
- ▶ Estar em uma **ordem crescente** ou **decrescente**

- ▶ **Exemplos:**

- ▶ **Entrada:** 6 5 7 1 2 4 3
- ▶ **Saída:** 1 2 3 4 5 6 7 (ordem crescente)

- ▶ **Entrada:** V U X Z Y
- ▶ **Saída:** U V X Y Z (ordem crescente)

Introdução

- ▶ A **saída de um algoritmo de ordenação** deve satisfazer duas condições:

- ▶ Ser uma **permutação da entrada**
- ▶ Estar em uma **ordem crescente ou decrescente**

- ▶ **Exemplos:**

- ▶ **Entrada:** 6 5 7 1 2 4 3

- ▶ **Saída:** 7 6 5 4 3 2 1 (ordem decrescente)

- ▶ **Entrada:** V U X Z Y

- ▶ **Saída:** Z Y X V U (ordem decrescente)

Classificação dos métodos de ordenação

▶ Quanto à **origem do arquivo**:

- ▶ **Ordenação interna:** dados e processo na memória principal
 - ▶ Acesso direto e rápido aos dados
- ▶ **Ordenação externa:** dados na memória secundária e processo na memória principal
 - ▶ Acesso sequencial ou em grandes blocos

Classificação dos métodos de ordenação

▶ Quanto à **origem do arquivo**:

- ▶ **Ordenação interna**: dados e processo na memória principal
 - ▶ Acesso direto e rápido aos dados
- ▶ **Ordenação externa**: dados na memória secundária e processo na memória principal
 - ▶ Acesso sequencial ou em grandes blocos

▶ Quanto à **estabilidade**:

- ▶ **Método estável**: algoritmo **preserva a ordem relativa original** dos registros com o mesmo valor de chave
- ▶ **Método não estável**: não preserva a ordem em que os registros aparecem no arquivo original

Exemplo

Arquivo original

Nome	Idade
Maria	20
José	60
João	20

método
estável

Nome	Idade
Maria	20
João	20
José	60

Arquivos ordenados

método
não estável

Nome	Idade
João	20
Maria	20
José	60

Classificação dos métodos de ordenação

▶ Quanto à **movimentação**:

- ▶ **Ordenação dos registros:** realiza a movimentação/cópia dos registros
- ▶ **Ordenação por ponteiros:** ordenação é feita sobre uma tabela auxiliar de ponteiros
 - ▶ **Não** movimenta os registros originais

Classificação dos métodos de ordenação

▶ Quanto à **movimentação**:

- ▶ **Ordenação dos registros**: realiza a movimentação/cópia dos registros
- ▶ **Ordenação por ponteiros**: ordenação é feita sobre uma tabela auxiliar de ponteiros
 - ▶ **Não** movimenta os registros originais

▶ Quanto à **complexidade**:

- ▶ **Algoritmos simples**: da ordem de $O(n^2)$
 - ▶ **Ex**: *bubble sort*
- ▶ **Algoritmos eficientes**: da ordem de $O(n \log n)$
 - ▶ **Ex**: *quick sort*

Classificação dos métodos de ordenação

- ▶ **Aspectos de eficiência:**

- ▶ Adequação da simplicidade/tamanho do problema com o método usado
 - ▶ Relacionado com tempo de execução ou memória necessária

Classificação dos métodos de ordenação

▶ Aspectos de eficiência:

- ▶ Adequação da simplicidade/tamanho do problema com o método usado
 - ▶ Relacionado com tempo de execução ou memória necessária
- ▶ Análise é feita pela contagem de **operações críticas**:
 - ▶ Comparações entre chaves
 - ▶ Movimentação de registros ou ponteiros

Classificação dos métodos de ordenação

▶ Aspectos de eficiência:

- ▶ Adequação da simplicidade/tamanho do problema com o método usado
 - ▶ Relacionado com tempo de execução ou memória necessária
- ▶ Análise é feita pela contagem de **operações críticas**:
 - ▶ Comparações entre chaves
 - ▶ Movimentação de registros ou ponteiros

▶ Questão: Por que estudar os algoritmos simples?

Classificação dos métodos de ordenação

▶ Aspectos de eficiência:

- ▶ Adequação da simplicidade/tamanho do problema com o método usado
 - ▶ Relacionado com tempo de execução ou memória necessária
- ▶ Análise é feita pela contagem de **operações críticas**:
 - ▶ Comparações entre chaves
 - ▶ Movimentação de registros ou ponteiros

▶ Questão: Por que estudar os algoritmos simples?

- ▶ Facilidade de implementação e entendimento
 - ▶ Ilustra com simplicidade os **princípios da ordenação por comparação**

Classificação dos métodos de ordenação

▶ Aspectos de eficiência:

- ▶ Adequação da simplicidade/tamanho do problema com o método usado
 - ▶ Relacionado com tempo de execução ou memória necessária
- ▶ Análise é feita pela contagem de **operações críticas**:
 - ▶ Comparações entre chaves
 - ▶ Movimentação de registros ou ponteiros

▶ Questão: Por que estudar os algoritmos simples?

- ▶ Facilidade de implementação e entendimento
 - ▶ Ilustra com simplicidade os **princípios da ordenação por comparação**
- ▶ Podem ser mais adequados em alguns casos
 - ▶ **Ex:** ordenação de **conjuntos pequenos**

Ordenação por "bolha" (*bubble sort*)

- ▶ Algoritmo de ordenação **simples**
- ▶ Não é recomendado para grandes volumes de dados
- ▶ **Ideia:**
 - ▶ Compara **pares de elementos adjacentes** (E_i e E_j)
 - ▶ Se $E_i > E_j$, **troca as suas posições**
 - ▶ Repete esse processo até o maior elemento estar no final do arranjo **ou não ocorrer mais trocas**

```
if (  $E[i] > E[i+1]$  ) {  
     $aux = E[i]$  ;  
     $E[i] = E[i+1]$  ;  
     $E[i+1] = aux$  ;  
}
```

Código para a
troca dos
elementos

Bubble sort : 1ª iteração

	0	1	2	3	4	5
início	3	1	6	2	8	4

Bubble sort : 1ª iteração

início

0	1	2	3	4	5
3	1	6	2	8	4
0	1	2	3	4	5
3	1	6	2	8	4

$E[i]$ $E[j]$ troca?

3 1 sim

Bubble sort : 1ª iteração

	0	1	2	3	4	5			
início	3	1	6	2	8	4	$E[i]$	$E[j]$	troca?
	0	1	2	3	4	5			
	3	1	6	2	8	4	3	1	sim
	0	1	2	3	4	5			
	1	3	6	2	8	4	3	6	não

Bubble sort : 1ª iteração

	0	1	2	3	4	5			
início	3	1	6	2	8	4	$E[i]$	$E[j]$	troca?
	0	1	2	3	4	5			
	3	1	6	2	8	4	3	1	sim
	0	1	2	3	4	5			
	1	3	6	2	8	4	3	6	não
	0	1	2	3	4	5			
	1	3	6	2	8	4	6	2	sim

Bubble sort : 1ª iteração

	0	1	2	3	4	5			
início	3	1	6	2	8	4	$E[i]$	$E[j]$	troca?
	0	1	2	3	4	5			
	3	1	6	2	8	4	3	1	sim
	0	1	2	3	4	5			
	1	3	6	2	8	4	3	6	não
	0	1	2	3	4	5			
	1	3	6	2	8	4	6	2	sim
	0	1	2	3	4	5			
	1	3	2	6	8	4	6	8	não

Bubble sort : 1ª iteração

	0	1	2	3	4	5			
início	3	1	6	2	8	4	$E[i]$	$E[j]$	troca?
	0	1	2	3	4	5			
	3	1	6	2	8	4	3	1	sim
	0	1	2	3	4	5			
	1	3	6	2	8	4	3	6	não
	0	1	2	3	4	5			
	1	3	6	2	8	4	6	2	sim
	0	1	2	3	4	5			
	1	3	2	6	8	4	6	8	não
	0	1	2	3	4	5			
	1	3	2	6	8	4	8	4	sim

Bubble sort : 1ª iteração

	0	1	2	3	4	5			
início	3	1	6	2	8	4	$E[i]$	$E[j]$	troca?
	0	1	2	3	4	5			
	3	1	6	2	8	4	3	1	sim
	0	1	2	3	4	5			
	1	3	6	2	8	4	3	6	não
	0	1	2	3	4	5			
	1	3	6	2	8	4	6	2	sim
	0	1	2	3	4	5			
	1	3	2	6	8	4	6	8	não
	0	1	2	3	4	5			
	1	3	2	6	8	4	8	4	sim
	0	1	2	3	4	5			
fim	1	3	2	6	4	8			

Bubble sort : 1ª iteração

	0	1	2	3	4	5			
início	3	1	6	2	8	4	$E[i]$	$E[j]$	troca?
	0	1	2	3	4	5			
	3	1	6	2	8	4	3	1	sim
	0	1	2	3	4	5			
	1	3	6	2	8	4	3	6	não
	0	1	2	3	4	5			
	1	3	6	2	8	4	6	2	sim
	0	1	2	3	4	5			
	1	3	2	6	8	4	6	8	não
	0	1	2	3	4	5			
	1	3	2	6	8	4	8	4	sim
	0	1	2	3	4	5			
fim	1	3	2	6	4	8			

maior elemento estará na sua posição ordenada

Bubble sort : outras iterações

início

0	1	2	3	4	5
3	1	6	2	8	4

Bubble sort : outras iterações

	0	1	2	3	4	5
início	3	1	6	2	8	4

	0	1	2	3	4	5
1ª iteração	1	3	2	6	4	8

Bubble sort : outras iterações

	0	1	2	3	4	5
início	3	1	6	2	8	4

	0	1	2	3	4	5
1ª iteração	1	3	2	6	4	8

	0	1	2	3	4	5
2ª iteração	1	2	3	4	6	8

Bubble sort : outras iterações

	0	1	2	3	4	5
início	3	1	6	2	8	4

	0	1	2	3	4	5
1ª iteração	1	3	2	6	4	8

	0	1	2	3	4	5
2ª iteração	1	2	3	4	6	8

	0	1	2	3	4	5
3ª iteração	1	2	3	4	6	8

Bubble sort : outras iterações

	0	1	2	3	4	5
início	3	1	6	2	8	4

	0	1	2	3	4	5
1ª iteração	1	3	2	6	4	8

	0	1	2	3	4	5
2ª iteração	1	2	3	4	6	8

	0	1	2	3	4	5
3ª iteração	1	2	3	4	6	8

⚡ troca (**dados estão ordenados**) - Fim do algoritmo

Bubble sort : implementação

```
void bubblesort (int vetor[], int n){
    int i, iteracao, aux;

    /*controle do número de iterações (n-1)*/
    for (iteracao = 0; iteracao < n-1; iteracao++)

        /*repeticao interna, percorrimento do vetor (n-1)*/
        for (i=0; i < n-1; i++)

            if (vetor[i] > vetor[i+1]) {
                /*Troca quando necessário*/
                aux = vetor[i];
                vetor[i] = vetor[i+1];
                vetor[i+1]=aux;
            }
}
```

Bubble sort : implementação

```
void bubblesort (int vetor[], int n){  
    int i, iteracao, aux;
```

n elementos

```
    /*controle do número de iterações (n-1)*/  
    for (iteracao = 0; iteracao < n-1; iteracao++)
```

```
    /*repeticao interna, percorrimo do vetor (n-1)*/  
    for (i=0; i < n-1; i++)
```

```
        if (vetor[i] > vetor[i+1]) {  
            /*Troca quando necessário*/  
            aux = vetor[i];  
            vetor[i] = vetor[i+1];  
            vetor[i+1]=aux;  
        }
```

```
}
```

Bubble sort : implementação

```
void bubblesort (int vetor[], int n){  
    int i, iteracao, aux;
```

n elementos

```
    /*controle do número de iterações (n-1)*/  
    for (iteracao = 0; iteracao < n-1; iteracao++)
```

$(n-1)$ iterações

```
    /*repeticao interna, percorrimo do vetor (n-1)*/  
    for (i=0; i < n-1; i++)
```

```
        if (vetor[i] > vetor[i+1]) {  
            /*Troca quando necessário*/  
            aux = vetor[i];  
            vetor[i] = vetor[i+1];  
            vetor[i+1]=aux;  
        }
```

```
}
```

Bubble sort : implementação

```
void bubblesort (int vetor[], int n){  
    int i, iteracao, aux;
```

n elementos

```
    /*controle do número de iterações (n-1)*/  
    for (iteracao = 0; iteracao < n-1; iteracao++)
```

$(n-1)$ iterações

```
    /*repeticao interna, percorrimento do vetor (n-1)*/  
    for (i=0; i < n-1; i++)
```

```
        if (vetor[i] > vetor[i+1]) {  
            /*Troca quando necessário*/  
            aux = vetor[i];  
            vetor[i] = vetor[i+1];  
            vetor[i+1]=aux;  
        }
```

$(n-1)$ comparações

```
}
```

Bubble sort : implementação

```
void bubblesort (int vetor[], int n){
    int i, iteracao, aux;

    /*controle do número de iterações (n-1)*/
    for (iteracao = 0; iteracao < n-1; iteracao++){

        /*repeticao interna, percorrimento do vetor (n-1)*/
        for (i=0; i < n-1; i++){

            if (vetor[i] > vetor[i+1]) {
                /*Troca quando necessário*/
                aux = vetor[i];
                vetor[i] = vetor[i+1];
                vetor[i+1]=aux;
            }
        }
    }
}
```

n elementos

$(n-1)$ iterações



$(n-1)$ comparações



$$g(n) = (n-1)^2$$

Bubble sort : implementação

```
void bubblesort (int vetor[], int n){  
    int i, iteracao, aux;
```

```
    /*controle do número de iterações (n-1)*/  
    for (iteracao = 0; iteracao < n-1; iteracao++)
```

```
    /*repeticao interna, percorrimto do vetor (n-1)*/  
    for (i=0; i < n-1; i++)
```

```
        if (vetor[i] > vetor[i+1]) {  
            /*Troca quando necessário*/  
            aux = vetor[i];  
            vetor[i] = vetor[i+1];  
            vetor[i+1]=aux;  
        }
```

Quantas trocas são realizadas?

```
}
```

Bubble sort : implementação

```
void bubblesort (int vetor[], int n){  
    int i, iteracao, aux;
```

```
    /*controle do número de iterações (n-1)*/  
    for (iteracao = 0; iteracao < n-1; iteracao++)
```

```
    /*repeticao interna, percorrimento do vetor (n-1)*/  
    for (i=0; i < n-1; i++)
```

```
    if (vetor[i] > vetor[i+1]) {  
        /*Troca quando necessário*/  
        aux = vetor[i];  
        vetor[i] = vetor[i+1];  
        vetor[i+1]=aux;  
    }
```

Quantas trocas são realizadas?

**Melhor caso (vetor ordenado):
nenhuma troca**

Bubble sort : implementação

```
void bubblesort (int vetor[], int n){
    int i, iteracao, aux;

    /*controle do número de iterações (n-1)*/
    for (iteracao = 0; iteracao < n-1; iteracao++)

        /*repeticao interna, percorrimento do vetor (n-1)*/
        for (i=0; i < n-1; i++)

            if (vetor[i] > vetor[i+1]) {
                /*Troca quando necessário*/
                aux = vetor[i];
                vetor[i] = vetor[i+1];
                vetor[i+1]=aux;
            }
}
```

$$\text{comparações}(n) = (n - 1)^2$$

$$\text{Melhor caso: trocas}(n) = 0$$

$$T(n) = (n - 1)^2 + 0 = n^2 - 2n + 1 = O(n^2)$$

Bubble sort : implementação

```
void bubblesort (int vetor[], int n){
    int i, iteracao, aux;

    /*controle do número de iterações (n-1)*/
    for (iteracao = 0; iteracao < n-1; iteracao++){

        /*repeticao interna, percorrimento do vetor (n-1)*/
        for (i=0; i < n-1; i++){

            if (vetor[i] > vetor[i+1]) {
                /*Troca quando necessário*/
                aux = vetor[i];
                vetor[i] = vetor[i+1];
                vetor[i+1]=aux;
            }
        }
    }
}
```

Iteração	Qtde. de trocas
1	$n-1$
2	$n-2$
3	$n-3$
4	$n-4$
...	
$n-1$	$n-(n-1) = 1$

Quantas trocas são realizadas?

Pior caso (ordem inversa):
 $(n(n-1)) / 2$

Bubble sort : implementação

```
void bubblesort (int vetor[], int n){  
    int i, iteracao, aux;  
  
    /*controle do número de iterações (n-1)*/  
    for (iteracao = 0; iteracao < n-1; iteracao++)  
  
        /*repeticao interna, percorrimento do vetor (n-1)*/  
        for (i=0; i < n-1; i++)  
  
            if (vetor[i] > vetor[i+1]) {  
                /*Troca quando necessário*/  
                aux = vetor[i];  
                vetor[i] = vetor[i+1];  
                vetor[i+1]=aux;  
            }  
}
```

$$\text{comparações}(n) = (n - 1)^2$$

$$\text{Pior caso: trocas}(n) = \frac{n(n-1)}{2}$$

$$T(n) = (n - 1)^2 + \frac{n(n - 1)}{2} = \frac{3n^2 - 5n + 2}{2} = O(n^2)$$

Bubble sort : implementação

- ▶ **Questões:**

- ▶ O método é estável?

Bubble sort : implementação

▶ **Questões:**

▶ **O método é estável?**

- ▶ Sim. A ordem relativa dos elementos iguais são mantidas.

Bubble sort : implementação

▶ **Questões:**

- ▶ **O método é estável?**
 - ▶ Sim. A ordem relativa dos elementos iguais são mantidas.
- ▶ **O algoritmo apresentado pode ser melhorado?**

Bubble sort : implementação

▶ Questões:

▶ O método é estável?

- ▶ Sim. A ordem relativa dos elementos iguais são mantidas.

▶ O algoritmo apresentado pode ser melhorado?

- ▶ Sim. Elementos já ordenados não precisam ser novamente comparados

...

```
for (iteracao = n-1; iteracao > 0; iteracao--)  
    for (i = 0; i < iteracao; i++)
```

...

Bubble sort : implementação

▶ Questões:

▶ O método é estável?

- ▶ Sim. A ordem relativa dos elementos iguais são mantidas.

▶ O algoritmo apresentado pode ser melhorado?

- ▶ Sim. Elementos já ordenados não precisam ser novamente comparados

...

for (iteracao = n-1; iteracao > 0; iteracao--)

for (i = 0; i < iteracao; i++)

...

- ▶ Se **não houver trocas** em uma iteração, o vetor já está ordenado e o algoritmo pode ser encerrado

Bubble sort : implementação

```
void bubblesort2 (int vetor[], int n){
    int i, iteracao, aux, troca;

    for (iteracao = n-1; iteracao > 0; iteracao--){

        troca = 0;
        for (i=0; i < iteracao; i++){
            if (vetor[i] > vetor[i+1]) {
                aux = vetor[i];
                vetor[i] = vetor[i+1];
                vetor[i+1]=aux;
                troca = 1;
            }

            if (troca == 0)
                break;
        }
    }
```

Bubble sort : implementação

```
void bubblesort2 (int vetor[], int n){  
    int i, iteracao, aux, troca;  
  
    for (iteracao = n-1; iteracao > 0; iteracao--)  
  
        troca = 0;  
        for (i=0; i < iteracao; i++)  
            if (vetor[i] > vetor[i+1]) {  
                aux = vetor[i];  
                vetor[i] = vetor[i+1];  
                vetor[i+1]=aux;  
                troca = 1;  
            }  
  
        if (troca == 0)  
            break;  
}
```

n elementos

Melhor caso: 1 iteração
Pior caso: (*n*-1) iterações



iteracao comparações
(varia a cada iteração)



Melhor caso: $g(n) = n-1 = \Omega(n)$
Pior caso: $g(n) = (n^2-n)/2 = O(n^2)$

Ordenação por "seleção" (*selection sort*)

- ▶ Um dos algoritmos mais **simples** de ordenação
- ▶ **Ideia:**
 - ▶ Selecione o **menor elemento** da região não ordenada
 - ▶ Troque-o com o 1º elemento dessa região
 - ▶ Repita esse processo até restar um único elemento na região não ordenada

Ordenação por "seleção" (*selection sort*)

- ▶ Um dos algoritmos mais **simples** de ordenação
- ▶ **Ideia:**
 - ▶ Selecione o **menor elemento** da região não ordenada
 - ▶ Troque-o com o 1º elemento dessa região
 - ▶ Repita esse processo até restar um único elemento na região não ordenada
- ▶ Recomendado para **pequenos conjuntos** de dados e para arquivos com **registros grandes**
 - ▶ Devido ao seu comportamento na troca de registros

Ordenação por "seleção" (*selection sort*)

- ▶ Um dos algoritmos mais **simples** de ordenação
- ▶ **Ideia:**
 - ▶ Selecione o **menor elemento** da região não ordenada
 - ▶ Troque-o com o 1º elemento dessa região
 - ▶ Repita esse processo até restar um único elemento na região não ordenada
- ▶ Recomendado para **pequenos conjuntos** de dados e para arquivos com **registros grandes**
 - ▶ Devido ao seu comportamento na troca de registros
- ▶ Não indicado para grandes conjuntos e **arquivos já ordenados**

Selection sort : exemplo de funcionamento



Iteração

Início 8 5 4 3 6

Selection sort : exemplo de funcionamento

Iteração

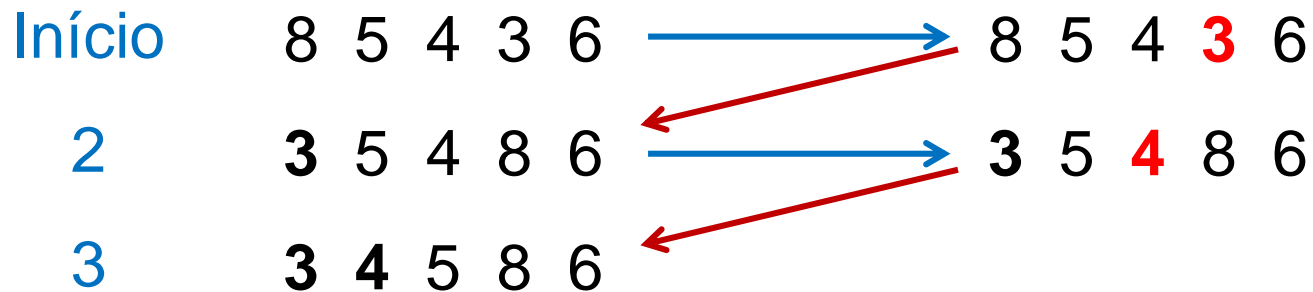
Menor elemento

Início	8	5	4	3	6		8	5	4	3	6
2	3	5	4	8	6						

Selection sort : exemplo de funcionamento

Iteração







Menor elemento



Selection sort : exemplo de funcionamento

Iteração









Menor elemento

Início	8 5 4 3 6		8 5 4 3 6
2	3 5 4 8 6	 	3 5 4 8 6
3	3 4 5 8 6	 	3 4 5 8 6
4	3 4 5 8 6		

Selection sort : exemplo de funcionamento

Iteração









Menor elemento

Início	8 5 4 3 6		8 5 4 3 6
2	3 5 4 8 6	 	3 5 4 8 6
3	3 4 5 8 6	 	3 4 5 8 6
4	3 4 5 8 6	 	3 4 5 8 6
5	3 4 5 6 8		

Selection sort : exemplo de funcionamento

Iteração

Menor elemento

Início	8 5 4 3 6		8 5 4 3 6
2	3 5 4 8 6	 	3 5 4 8 6
3	3 4 5 8 6	 	3 4 5 8 6
4	3 4 5 8 6	 	3 4 5 8 6
5	3 4 5 6 8		
Fim	3 4 5 6 8		(vetor ordenado)

Selection sort : implementação

```
void selectionsort (int vetor[], int n) {  
    int i, iteracao, aux, menor;  
    // Controle do número de iterações  
    for (iteracao = 0; iteracao < n-1; iteracao++) {  
  
        // Busca o menor elemento  
        menor = iteracao;  
        for (i=iteracao+1; i < n; i++)  
            if (vetor[i] < vetor[menor])  
                menor = i;  
  
        // Troca os elementos  
        if (iteracao != menor) {  
            aux = vetor[iteracao];  
            vetor[iteracao] = vetor[menor];  
            vetor[menor] = aux;  
        }  
    }  
}
```

Selection sort : implementação

```
void selectionsort (int vetor[], int n) {  
    int i, iteracao, aux, menor;  
    // Controle do número de iterações  
    for (iteracao = 0; iteracao < n-1; iteracao++) {  
  
        // Busca o menor elemento  
        menor = iteracao;  
        for (i=iteracao+1; i < n; i++)  
            if (vetor[i] < vetor[menor])  
                menor = i;  
  
        // Troca os elementos  
        if (iteracao != menor) {  
            aux = vetor[iteracao];  
            vetor[iteracao] = vetor[menor];  
            vetor[menor] = aux;  
        }  
    }  
}
```

Propriedades:

- ▶ Ordenação **NÃO** estável
- ▶ **$O(n^2)$** para comparações: $g(n) = (n^2 - n)/2$
- ▶ **$O(n)$** para trocas: $g(n) = n$
- ▶ **$O(1)$** para espaço extra

Ordenação por "inserção" (*insertion sort*)

- ▶ Algoritmo de ordenação **simples** baseado na **organização de cartas de baralho** na mão

Ordenação por "inserção" (*insertion sort*)

- ▶ Algoritmo de ordenação **simples** baseado na **organização de cartas de baralho** na mão
- ▶ **Ideia:**
 - ▶ Selecione um elemento por vez da **região não ordenada**
 - ▶ Coloque-o na **posição correta** em relação aos **elementos já ordenados**
 - ▶ Repita esse processo para todos os elementos

Ordenação por "inserção" (*insertion sort*)

- ▶ Algoritmo de ordenação **simples** baseado na **organização de cartas de baralho** na mão
- ▶ **Ideia:**
 - ▶ Selecione um elemento por vez da **região não ordenada**
 - ▶ Coloque-o na **posição correta** em relação aos **elementos já ordenados**
 - ▶ Repita esse processo para todos os elementos
- ▶ **Indicado para:**
 - ▶ Conjuntos **pequenos** de dados
 - ▶ Arquivos **quase ordenados**
 - ▶ Devido à quantidade de comparações exigidas

Insertion sort : exemplo de funcionamento

► Analogia com as cartas de baralho:

Distribuição das cartas:

Recebe a carta 10

Organização na mão:

10

Insertion sort : exemplo de funcionamento

► Analogia com as cartas de baralho:

Distribuição das cartas:

Recebe a carta 10

Recebe a carta 5

Organização na mão:

10

5 10

Insertion sort : exemplo de funcionamento

► Analogia com as cartas de baralho:

Distribuição das cartas:

Recebe a carta 10

Recebe a carta 5

Recebe a carta 4

Organização na mão:

10

5 10

4 5 10

Insertion sort : exemplo de funcionamento

► Analogia com as cartas de baralho:

Distribuição das cartas:

Recebe a carta 10

Recebe a carta 5

Recebe a carta 4

Recebe a carta 3

Organização na mão:

10

5 10

4 5 10

3 4 5 10

Insertion sort : exemplo de funcionamento

► Analogia com as cartas de baralho:

Distribuição das cartas:

Recebe a carta 10

Recebe a carta 5

Recebe a carta 4

Recebe a carta 3

Recebe a carta 6

Organização na mão:

10

5 10

4 5 10

3 4 5 10

3 4 5 **6** 10

Insertion sort : exemplo de funcionamento

► Analogia com as cartas de baralho:

Distribuição das cartas:

Recebe a carta 15

Recebe a carta 5

Recebe a carta 4

Recebe a carta 3

Recebe a carta 6

Organização na mão:

10

5 10

4 5 10

3 4 5 10

3 4 5 **6** 10

Questão: como fazer esse procedimento em um vetor dado que **não há uma visão global** dos elementos?

Insertion sort : exemplo de funcionamento

- ▶ **Solução:** comparar com os elementos já ordenados
 - ▶ Percorrer na **ordem contrária** (da direita para a esquerda)
 - ▶ Garante **eficiência** (para sem verificar todos) e **estabilidade**

Insertion sort : exemplo de funcionamento

- ▶ **Solução:** comparar com os elementos já ordenados
 - ▶ Percorrer na **ordem contrária** (da direita para a esquerda)
 - ▶ Garante **eficiência** (para sem verificar todos) e **estabilidade**

Iteração

Organização

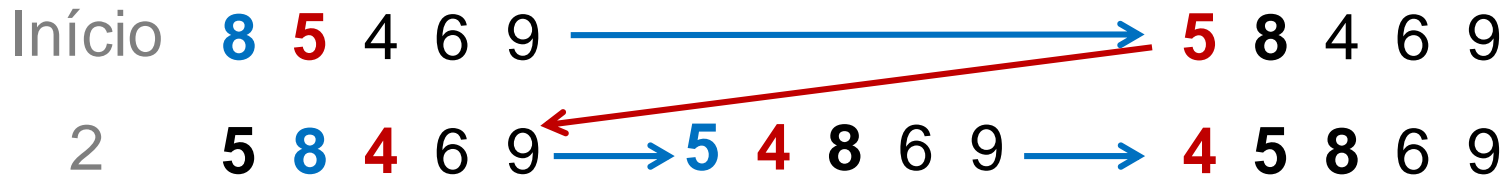
Início 8 5 4 6 9  5 8 4 6 9

Insertion sort : exemplo de funcionamento

- ▶ **Solução:** comparar com os elementos já ordenados
 - ▶ Percorrer na **ordem contrária** (da direita para a esquerda)
 - ▶ Garante **eficiência** (para sem verificar todos) e **estabilidade**

Iteração

Organização

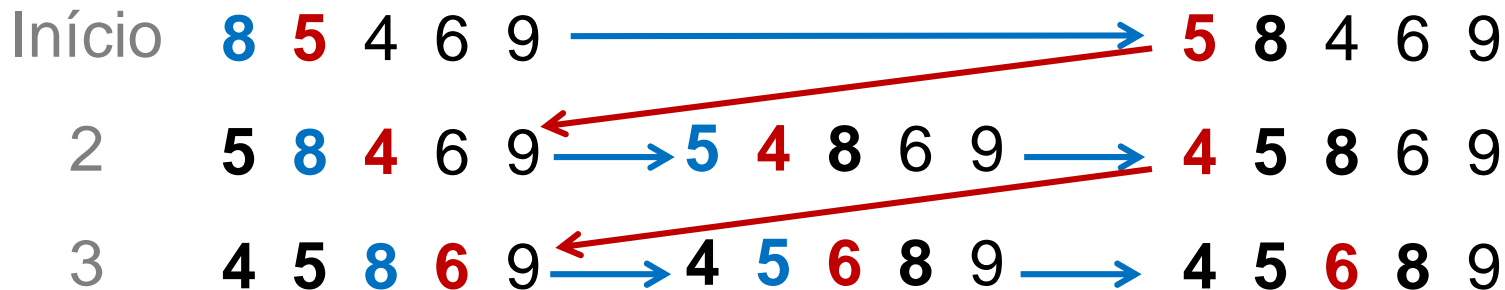


Insertion sort : exemplo de funcionamento

- ▶ **Solução:** comparar com os elementos já ordenados
 - ▶ Percorrer na **ordem contrária** (da direita para a esquerda)
 - ▶ Garante **eficiência** (para sem verificar todos) e **estabilidade**

Iteração

Organização

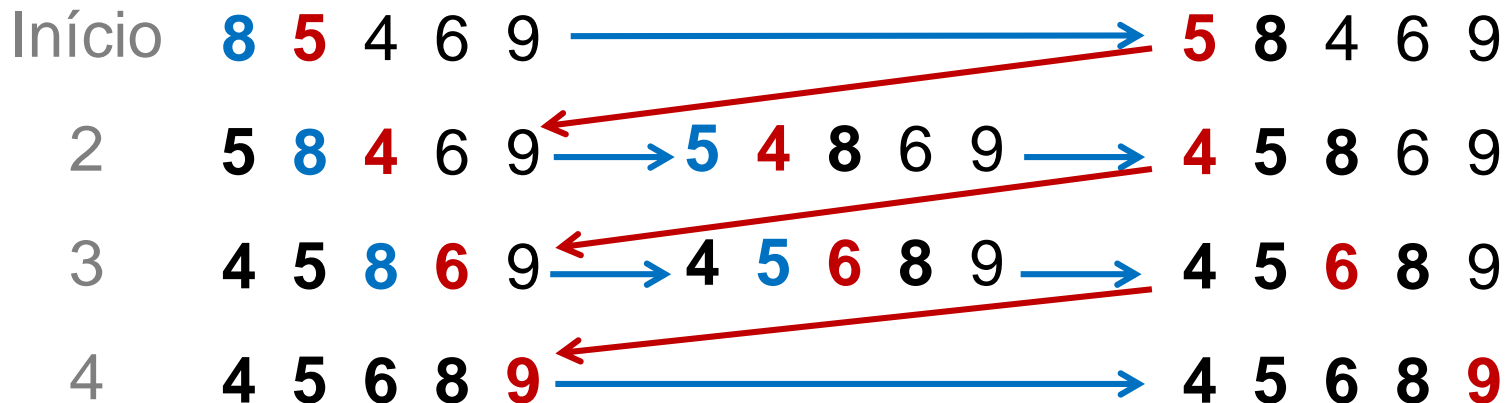


Insertion sort : exemplo de funcionamento

- ▶ **Solução:** comparar com os elementos já ordenados
 - ▶ Percorrer na **ordem contrária** (da direita para a esquerda)
 - ▶ Garante **eficiência** (para sem verificar todos) e **estabilidade**

Iteração

Organização

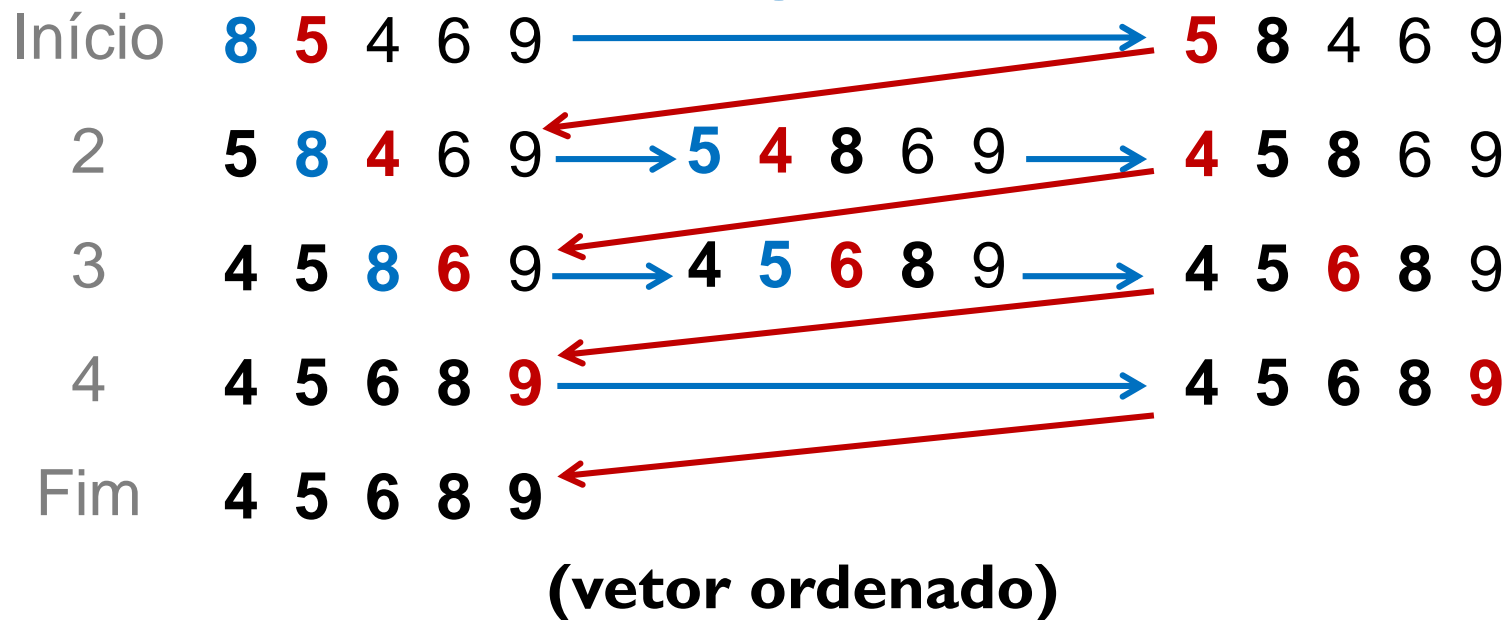


Insertion sort : exemplo de funcionamento

- ▶ **Solução:** comparar com os elementos já ordenados
 - ▶ Percorrer na **ordem contrária** (da direita para a esquerda)
 - ▶ Garante **eficiência** (para sem verificar todos) e **estabilidade**

Iteração

Organização



Insertion sort : implementação

```
void insertionsort (int vetor[], int n) {  
    int i, iteracao, elem;  
    // Controle do número de iterações  
    for (iteracao = 1; iteracao < n; iteracao++) {  
  
        // Busca posicao do elemento  
        elem = vetor[iteracao];  
        i = iteracao-1;  
        while (i >= 0 && vetor[i] > elem) {  
            vetor[i+1] = vetor[i];  
            i--;  
        }  
        // Posiciona elemento  
        vetor[i+1] = elem;  
    }  
}
```

Insertion sort : implementação

```
void insertionsort (int vetor[], int n) {  
    int i, iteracao, elem;  
    // Controle do número de iterações  
    for (iteracao = 1; iteracao < n; iteracao++) {  
  
        // Busca posicao do elemento  
        elem = vetor[iteracao];  
        i = iteracao-1;  
        while (i >= 0 && vetor[i] > elem) {  
            vetor[i+1] = vetor[i];  
            i--;  
        }  
        // Posiciona elemento  
        vetor[i+1] = elem;  
    }  
}
```

n elementos

(*n*-1) iterações



Melhor caso: 1 comparação
Pior caso: (*n*-1) comparações



Melhor caso: $g(n) = n-1 = \Omega(n)$
Pior caso: $g(n) = (n^2-n)/2 = O(n^2)$

Ordenação "rápida" (*quick sort*)

- ▶ Algoritmo de ordenação **eficiente** que utiliza a estratégia "**dividir para conquistar**"

Ordenação "rápida" (*quick sort*)

- ▶ Algoritmo de ordenação **eficiente** que utiliza a estratégia "**dividir para conquistar**"
- ▶ **Ideia:** núcleo do método está na **partição** de uma lista não ordenada
 - ▶ A partição rearranja os elementos de uma lista $L[1...n]$ e devolve um índice $i \in \{1...n\}$, tal que:

$$L[1...i-1] \leq L[i] \leq L[i+1...n]$$

Ordenação "rápida" (*quick sort*)

- ▶ Algoritmo de ordenação **eficiente** que utiliza a estratégia "**dividir para conquistar**"
- ▶ **Ideia:** núcleo do método está na **partição** de uma lista não ordenada
 - ▶ A partição rearranja os elementos de uma lista $L[1...n]$ e devolve um índice $i \in \{1...n\}$, tal que:

$$L[1...i-1] \leq L[i] \leq L[i+1...n]$$

- ▶ O elemento $v = L[i]$ é chamado de **pivô**

Ordenação "rápida" (*quick sort*)

- ▶ **Vantagem:** excelente desempenho
 - ▶ Forma **mais rápida ordenação** baseada em comparações de arranjos
 - ▶ Se bem implementado, executa quase sempre em $\theta(n \log n)$
 - ▶ No pior caso pode executar em tempo $O(n^2)$

Ordenação "rápida" (*quick sort*)

- ▶ **Vantagem:** excelente desempenho

- ▶ Forma **mais rápida ordenação** baseada em comparações de arranjos
 - ▶ Se bem implementado, executa quase sempre em $\theta(n \log n)$
 - ▶ No pior caso pode executar em tempo $O(n^2)$

- ▶ **Desvantagens:**

- ▶ Implementação **recursiva**
- ▶ **Não é estável**
- ▶ Ineficiente para listas ordenadas ou quando pivô é mal escolhido
 - ▶ Partições extremamente desiguais

Ordenação "rápida" (*quick sort*)

► **Algoritmo:**

- I. Iniciar com uma lista L de n itens

Ordenação "rápida" (*quick sort*)

► **Algoritmo:**

1. Iniciar com uma lista L de n itens
2. Escolher um item pivô v , dentre os elementos de L

Ordenação "rápida" (*quick sort*)

► Algoritmo:

1. Iniciar com uma lista L de n itens
2. Escolher um item pivô v , dentre os elementos de L
3. Particionar L em duas listas não ordenadas: $L1$ e $L2$
 - $L1$: conterá todas as chaves menores que v
 - $L2$: conterá todas as chaves maiores que v
 - Elementos iguais a v podem fazer parte de $L1$ ou $L2$
 - O pivô v não faz parte de nenhuma das duas listas

Ordenação "rápida" (*quick sort*)

► Algoritmo:

1. Iniciar com uma lista L de n itens
2. Escolher um item pivô v , dentre os elementos de L
3. Particionar L em duas listas não ordenadas: $L1$ e $L2$
 - $L1$: conterá todas as chaves menores que v
 - $L2$: conterá todas as chaves maiores que v
 - Elementos iguais a v podem fazer parte de $L1$ ou $L2$
 - O pivô v não faz parte de nenhuma das duas listas
4. Ordenar:
 1. $L1$ recursivamente, obtendo a lista ordenada $S1$
 2. $L2$ recursivamente, obtendo a lista ordenada $S2$

Ordenação "rápida" (*quick sort*)

► Algoritmo:

1. Iniciar com uma lista L de n itens
2. Escolher um item pivô v , dentre os elementos de L
3. Particionar L em duas listas não ordenadas: $L1$ e $L2$
 - $L1$: conterá todas as chaves menores que v
 - $L2$: conterá todas as chaves maiores que v
 - Elementos iguais a v podem fazer parte de $L1$ ou $L2$
 - O pivô v não faz parte de nenhuma das duas listas
4. Ordenar:
 1. $L1$ recursivamente, obtendo a lista ordenada $S1$
 2. $L2$ recursivamente, obtendo a lista ordenada $S2$
5. Concatenar $S1, v, S2$ produzindo a lista ordenada S

Quick sort: exemplo 1

- ▶ Considerando o pivô como o 1º elemento da lista
- ▶ Na fase de partição formaremos duas sub-listas: $L1$ e $L2$

	4	7	1	5	9	3	0
$L1$	1	3	0				
$L2$	7	5	9				



Quick sort: exemplo 1

- ▶ Considerando o pivô como o 1º elemento da lista
- ▶ Na fase de partição formaremos duas sub-listas: $L1$ e $L2$

	4	7	1	5	9	3	0
$L1$	1	3	0				
$L2$	7	5	9				

- ▶ $L1$ é particionada recursivamente
- ▶ Como alcançamos o **caso base**, as sub-listas são concatenadas

	1	3	0
$L1.1$	0		
$L1.2$	3		
$S1$	0	1	3



Quick sort: exemplo 1

	4	7	1	5	9	3	0
<i>SI</i>	0	1	3				
<i>L2</i>	7	5	9				



Quick sort: exemplo 1

	4	7	1	5	9	3	0
<i>S1</i>	0	1	3				
<i>L2</i>	7	5	9				

- ▶ *L2* é particionada recursivamente
- ▶ Como alcançamos o **caso base**, as sub-listas são concatenadas

	7	5	9
<i>L2.1</i>	5		
<i>L2.2</i>	9		
<i>S2</i>	5	7	9



Quick sort: exemplo 1

	4	7	1	5	9	3	0
S1	0	1	3				
S2	5	7	9				

- ▶ As sub-listas retornadas em cada iteração recursiva são concatenadas até obter a lista ordenada S

0	1	3	4	5	7	9
---	---	---	---	---	---	---



Quick sort: exemplo 2

- ▶ Considere um arranjo de números ordenados:

0	1	3	4	5	7	9
---	---	---	---	---	---	---

- ▶ Qual é o custo em tempo de execução do algoritmo?



Quick sort: exemplo 2

- ▶ Considere um arranjo de números ordenados:

0	1	3	4	5	7	9
---	---	---	---	---	---	---

- ▶ Qual é o custo em tempo de execução do algoritmo?
 - ▶ **$O(n^2)$**
 - ▶ Neste caso, usar o 1º elemento como pivô não é uma boa estratégia

	0	1	3	4	5	7	9
L1							
L2	1	3	4	5	7	9	



Quick sort: escolha do pivô

- ▶ A escolha do pivô afeta significativamente o desempenho do algoritmo
 - ▶ Fase de **partição** é a parte **crítica**
- ▶ Existem várias estratégias possíveis:
 - ▶ 1º elemento
 - ▶ Elemento do meio
 - ▶ Elemento mais próximo da média
 - ▶ Mediana
 - ▶ Entre outros



Quick sort: exemplo 3

- Uso do elemento do meio como pivô:

	1	2	3	4	3	2	1
L1	1	2	3	3	2	1	
L2							



Quick sort: exemplo 3

- Uso do elemento do meio como pivô:

	1	2	3	4	3	2	1
L1	1	2	3	3	2	1	
L2							

	1	2	3	3	2	1
L1.1	1	2	2	1		
L1.2	3					



Quick sort: exemplo 3

- Uso do elemento do meio como pivô:

	1	2	3	4	3	2	1
L1	1	2	3	3	2	1	
L2							

	1	2	3	3	2	1
L1.1	1	2	2	1		
L1.2	3					

	1	2	2	1
L1.1.1	1	1		
L1.1.2	2			



Quick sort: exemplo 3

- Uso do elemento do meio como pivô:

	1	2	3	4	3	2	1
L1	1	2	3	3	2	1	
L2							

	1	2	3	3	2	1
L1.1	1	2	2	1		
L1.2	3					

	1	2	2	1
L1.1.1	1	1		
L1.1.2	2			

	1	1
L1.1.1.1	1	
L1.1.1.2		



Quick sort: exemplo 3

- Uso do elemento do meio como pivô:

	I	I
<i>LI.I.I.I</i>	I	
<i>LI.I.I.2</i>		
<i>SI.I.I</i>	I	I



Quick sort: exemplo 3

- Uso do elemento do meio como pivô:

	1	1
<i>L1.1.1.1</i>	1	
<i>L1.1.1.2</i>		
<i>S1.1.1</i>	1	1

	1	2	2	1
<i>S1.1.1</i>	1	1		
<i>S1.1.2</i>	2			
<i>S1.1</i>	1	1	2	2



Quick sort: exemplo 3

- Uso do elemento do meio como pivô:

	1	1
L1.1.1.1	1	
L1.1.1.2		
S1.1.1	1	1

	1	2	2	1
S1.1.1	1	1		
S1.1.2	2			
S1.1	1	1	2	2

	1	2	3	3	2	1
S1.1	1	1	2	2		
S1.2	3					
S1	1	1	2	2	3	3



Quick sort: exemplo 3

- Uso do elemento do meio como pivô:

	1	2	3	3	2	1
<i>SI.1</i>	1	1	2	2		
<i>SI.2</i>	3					
<i>SI</i>	1	1	2	2	3	3

	1	2	3	4	3	2	1
<i>S1</i>	1	1	2	2	3	3	
<i>S2</i>							
<i>S</i>	1	1	2	2	3	3	4

Quick sort: escolha do pivô

- ▶ Para a escolha do pivô mais adequado é necessário conhecer a **distribuição dos dados**
 - ▶ Usa o elemento mais adequado à distribuição



Quick sort: escolha do pivô

- ▶ Para a escolha do pivô mais adequado é necessário conhecer a **distribuição dos dados**
 - ▶ Usa o elemento mais adequado à distribuição
- ▶ Se a distribuição não é conhecida, a escolha deve ser **aleatória**
 - ▶ Na média, gera uma partição na proporção $1/4$ e $3/4$
 - ▶ Se essa proporção ocorrer em **metade das partições**, o tempo de execução esperado é **$\theta(n \log n)$**



Quick sort: escolha do pivô

- ▶ **Mediana de três:**

- ▶ Estratégia aleatória usada para aumentar as chances de obter o custo $\theta(n \log n)$



Quick sort: escolha do pivô

▶ **Mediana de três:**

- ▶ Estratégia aleatória usada para aumentar as chances de obter o custo **$O(n \log n)$**
- ▶ **Ideia:** escolher 3 elementos aleatórios e adotar como pivô o elemento central (mediana entre os 3)



Quick sort: escolha do pivô

▶ **Mediana de três:**

- ▶ Estratégia aleatória usada para aumentar as chances de obter o custo **$O(n \log n)$**
- ▶ **Ideia:** escolher 3 elementos aleatórios e adotar como pivô o elemento central (mediana entre os 3)
- ▶ Indicada apenas na **ordenação de listas grandes**
 - ▶ Para listas pequenas deve-se usar a escolha aleatória simples
 - O custo da estratégia não compensa



Quick sort em listas encadeadas

- ▶ Vantajoso tratar o problema como a **partição em 3 listas**:
 - ▶ $L1$ contendo chaves **menores** que o pivô
 - ▶ $L2$ contendo chaves **maiores** que o pivô
 - ▶ Lv contendo chaves **iguais** ao pivô
- ▶ Ordenação é feita apenas em $L1$ e $L2$
 - ▶ Lv não precisa ser ordenado
- ▶ A concatenação é realizada na forma: $S1 \rightarrow Lv \rightarrow S2$

	5	7	5	0	6	5	5
$L1$	0						
$L2$	7	6					
Lv	5	5	5	5			

Quick sort em arranjos (listas sequenciais)

- ▶ Algoritmo realiza ordenação ***in-place***
 - ▶ Utiliza **movimentações** dentro do próprio arranjo
 - ▶ **NÃO** usa de memória auxiliar
- ▶ Deve-se considerar as características do problema para evitar casos de execução quadrática
 - ▶ Mesmo algoritmos de livros podem ser lentos



Quick sort em arranjos

- ▶ **Problema:** Dado um arranjo A , ordene os itens de $A[p]$ até $A[r]$
- ▶ **Algoritmo:**
 - ▶ Escolha um pivô v e substitua-o pelo último item ($A[r]$)
 - ▶ Crie 2 variáveis de controle: $i = p-1$ e $j = r$

3	8	4	0	9	7	5
p		v				r
3	8	5	0	9	7	4
i						j

- ▶ O arranjo será ordenado para as posições maiores que i e menores que j



Quicksort em arranjos: algoritmo

3	8	5	0	9	7	4
i						j

▶ Invariantes:

- ▶ Elementos à esquerda de i são menores ou iguais ao pivô
- ▶ Elementos à direita de j são maiores ou iguais ao pivô

▶ Operações:

- ▶ Incrementar i até encontrar chave maior ou igual ao pivô
- ▶ Decrementar j até encontrar chave menor ou igual ao pivô
- ▶ Trocar itens $A[i]$ e $A[j]$
- ▶ Parar quando $i \geq j$
- ▶ Substituir o pivô com o elemento na posição i



Quick sort : implementação

```
int quicksort(int a[], int p, int r) {  
    int t;  
    if (p < r) {  
        int v = (rand()%(r-p))+p;  
        int pivo = a[v]; a[v] = a[r]; a[r] = pivo; // Opcional  
        int i = p-1; int j = r;  
  
        do {  
            do { i++; } while (a[i] < pivo);  
            do { j--; } while ((a[j] > pivo) && (j > p));  
            if (i < j) {t = a[i]; a[i] = a[j]; a[j] = t;} // troca i com j  
        } while (i < j);  
        a[r] = a[i]; a[i] = pivo; // Opcional  
  
        // chamadas recursivas  
        quicksort(a, p, i-1); quicksort(a, i+1, r);  
    }  
}
```

Quick sort: análise do algoritmo

- ▶ O desempenho do algoritmo está relacionado como a **divisão em subproblemas**
- ▶ **Pior caso**: gera-se uma partição de tamanho $n-1$ e outra de tamanho 0 em todas as chamadas recursivas
- ▶ **Melhor caso**: o problema é dividido ao meio, ou seja, uma partição tem tamanho $\text{floor}(n/2)$ e outra $\text{ceil}(n/2)-1$
- ▶ **Caso médio**: se aproxima do melhor caso

Quick sort: análise do algoritmo

```
void quicksort(int a[], int p, int r) {  
    int t;  
    if (p < r) {  
        int v = (rand()%(r-p))+p; // escolhe pivo aleatoriamente  
        int pivo = a[v];  
        a[v] = a[r]; a[r] = pivo; // troca pivo e ultimo elemento  
        int i = p-1; int j = r;  
        do {  
            do { i++; } while (a[i] < pivo);  
            do { j--; } while ((a[j] > pivo) && (j > p));  
            if (i < j) {  
                t = a[i], a[i] = a[j], a[j] = t; // troca i com j  
            }  
        } while (i < j);  
    }
```

Processo de
divisão do vetor
percorre todo
vetor $\theta(n)$

```
        a[r] = a[i], a[i] = pivo;  
        quicksort(a, p, i-1);  
        quicksort(a, i+1, r);  
    }  
}
```

Quick sort: análise do algoritmo

```
void quicksort(int a[], int p, int r) {  
    int t;  
    if (p < r) {  
        int v = (rand()%(r-p))+p; // escolhe pivo aleatoriamente  
        int pivo = a[v];  
        a[v] = a[r]; a[r] = pivo; // troca pivo e ultimo elemento  
        int i = p-1; int j = r;
```

```
        do {  
            do { i++; } while (a[i] < pivo);  
            do { j--; } while ((a[j] > pivo) && (j > p));  
            if (i < j) {  
                t = a[i], a[i] = a[j], a[j] = t; // troca i com j  
            }  
        } while (i < j);
```

Processo de
divisão do vetor
percorre todo
vetor $\theta(n)$



```
        a[r] = a[i], a[i] = pivo;  
        quicksort(a, p, i-1);  
        quicksort(a, i+1, r);
```

Tempo da 1ª parte

```
    }
```

```
}
```

Quick sort: análise do algoritmo

```
void quicksort(int a[], int p, int r) {  
    int t;  
    if (p < r) {  
        int v = (rand()%(r-p))+p; // escolhe pivo aleatoriamente  
        int pivo = a[v];  
        a[v] = a[r]; a[r] = pivo; // troca pivo e ultimo elemento  
        int i = p-1; int j = r;
```

```
        do {  
            do { i++; } while (a[i] < pivo);  
            do { j--; } while ((a[j] > pivo) && (j > p));  
            if (i < j) {  
                t = a[i], a[i] = a[j], a[j] = t; // troca i com j  
            }  
        } while (i < j);
```

Processo de
divisão do vetor
percorre todo
vetor $\theta(n)$



```
        a[r] = a[i], a[i] = pivo;
```

```
        quicksort(a, p, i-1);
```

```
        quicksort(a, i+1, r);  
    }
```

Tempo da 1ª parte



Tempo da 2ª parte

Quick sort: análise do algoritmo

```
void quicksort(int a[], int p, int r) {  
    int t;  
    if (p < r) {  
        int v = (rand()%(r-p))+p; // escolhe pivo aleatoriamente  
        int pivo = a[v];  
        a[v] = a[r]; a[r] = pivo; // troca pivo e ultimo elemento  
        int i = p-1; int j = r;
```

Pior
caso



```
    do {  
        do { i++; } while (a[i] < pivo);  
        do { j--; } while ((a[j] > pivo) && (j > p));  
        if (i < j) {  
            t = a[i], a[i] = a[j], a[j] = t; // troca i com j  
        }  
    } while (i < j);
```

Processo de
divisão do vetor
percorre todo
vetor $\theta(n)$



```
    a[r] = a[i], a[i] = pivo;
```

```
    quicksort(a, p, i-1);
```

```
    quicksort(a, i+1, r);  
}
```

primeira parte: $T(n-1)$



segunda parte: 0

Quick sort: análise do algoritmo

```
void quicksort(int a[], int p, int r) {  
    int t;  
    if (p < r) {  
        int v = (rand()%(r-p))+p; // escolhe pivo aleatoriamente  
        int pivo = a[v];  
        a[v] = a[r]; a[r] = pivo; // troca pivo e ultimo elemento  
        int i = p-1; int j = r;
```

```
        do {  
            do { i++; } while (a[i] < pivo);  
            do { j--; } while ((a[j] > pivo) && (j > p));  
            if (i < j) {  
                t = a[i], a[i] = a[j], a[j] = t; // troca i com j  
            }  
        } while (i < j);
```

```
        a[r] = a[i], a[i] = pivo;
```

```
        quicksort(a, p, i-1);
```

```
        quicksort(a, i+1, r);  
    }
```

Pior
caso

$$T(n) = T(n-1) + \theta(n) \\ \approx O(n^2)$$

Processo de
divisão do vetor
percorre todo
vetor $\theta(n)$



primeira parte: $T(n-1)$




segunda parte: 0

Quick sort: análise do algoritmo

```
void quicksort(int a[], int p, int r) {  
    int t;  
    if (p < r) {  
        int v = (rand()%(r-p))+p; // escolhe pivo aleatoriamente  
        int pivo = a[v];  
        a[v] = a[r]; a[r] = pivo; // troca pivo e ultimo elemento  
        int i = p-1; int j = r;
```

Melhor
caso



```
    do {  
        do { i++; } while (a[i] < pivo);  
        do { j--; } while ((a[j] > pivo) && (j > p));  
        if (i < j) {  
            t = a[i], a[i] = a[j], a[j] = t; // troca i com j  
        }  
    } while (i < j);
```

Processo de
divisão do vetor
percorre todo
vetor $\theta(n)$



```
    a[r] = a[i], a[i] = pivo;
```

```
    quicksort(a, p, i-1);
```

```
    quicksort(a, i+1, r);  
}
```

primeira parte: $T(n/2)$



segunda parte: $T(n/2)$

Quick sort: análise do algoritmo

```
void quicksort(int a[], int p, int r) {  
    int t;  
    if (p < r) {  
        int v = (rand()%(r-p))+p; // escolhe pivo aleatoriamente  
        int pivo = a[v];  
        a[v] = a[r]; a[r] = pivo; // troca pivo e ultimo elemento  
        int i = p-1; int j = r;
```

Melhor
caso



$$T(n) = 2T(n/2) + \theta(n) \\ \approx O(n \log n)$$

```
    do {  
        do { i++; } while (a[i] < pivo);  
        do { j--; } while ((a[j] > pivo) && (j > p));  
        if (i < j) {  
            t = a[i], a[i] = a[j], a[j] = t; // troca i com j  
        }  
    } while (i < j);
```

Processo de
divisão do vetor
percorre todo
vetor $\theta(n)$



```
    a[r] = a[i], a[i] = pivo;
```

```
    quicksort(a, p, i-1);
```

```
    quicksort(a, i+1, r);  
}
```

primeira parte: $T(n/2)$



segunda parte: $T(n/2)$

OUTROS ALGORITMOS



Ordenação por "mistura" (*merge sort*)

- ▶ Ordenação também baseada no "**dividir para conquistar**"
- ▶ **Ideia:**
 - ▶ **Particiona repetidamente o conjunto de dados** até que cada subconjunto tenha apenas 1 elemento
 - ▶ **Intercala duas partições** menores a fim de obter um subconjunto maior e ordenado, até restar um único conjunto

Ordenação por "mistura" (*merge sort*)

- ▶ Ordenação também baseada no "**dividir para conquistar**"
- ▶ **Ideia:**
 - ▶ **Particiona repetidamente o conjunto de dados** até que cada subconjunto tenha apenas 1 elemento
 - ▶ **Intercala duas partições** menores a fim de obter um subconjunto maior e ordenado, até restar um único conjunto
- ▶ **Vantagem:**
 - ▶ Excelente desempenho: **$O(n \log n)$** no melhor e pior casos

Ordenação por "mistura" (*merge sort*)

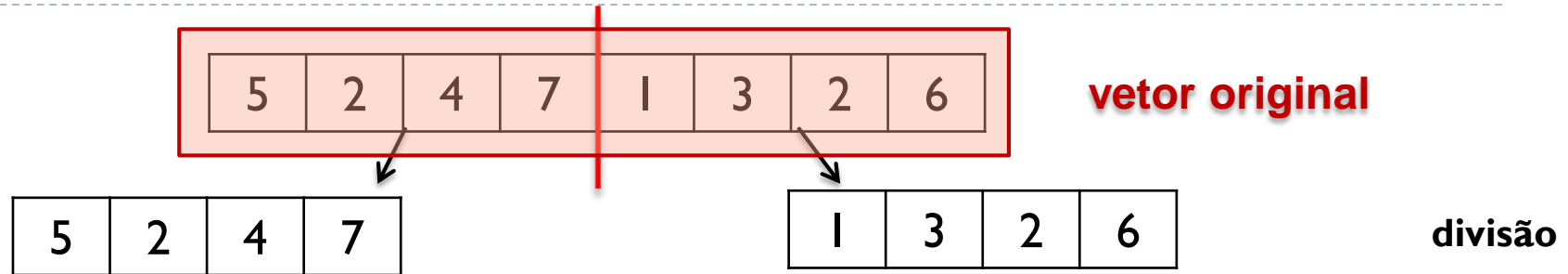
- ▶ Ordenação também baseada no "**dividir para conquistar**"
- ▶ **Ideia:**
 - ▶ **Particiona repetidamente o conjunto de dados** até que cada subconjunto tenha apenas 1 elemento
 - ▶ **Intercala duas partições** menores a fim de obter um subconjunto maior e ordenado, até restar um único conjunto
- ▶ **Vantagem:**
 - ▶ Excelente desempenho: **$O(n \log n)$** no melhor e pior casos
- ▶ **Desvantagens:**
 - ▶ Implementação **recursiva**
 - ▶ Precisa de vetor auxiliar (> **custo de memória**)

Merge sort : exemplo de funcionamento

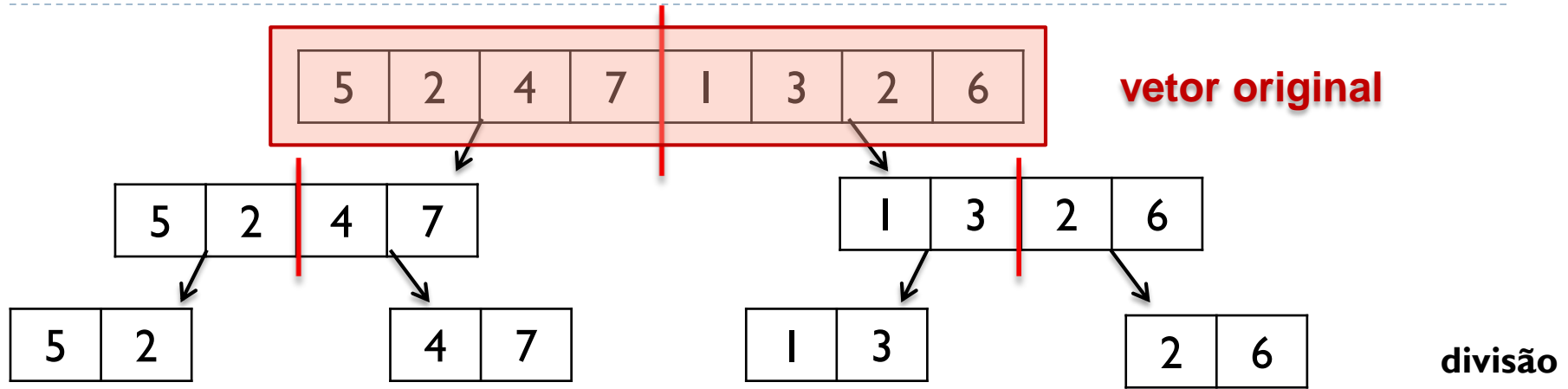
5	2	4	7	1	3	2	6
---	---	---	---	---	---	---	---

vetor original

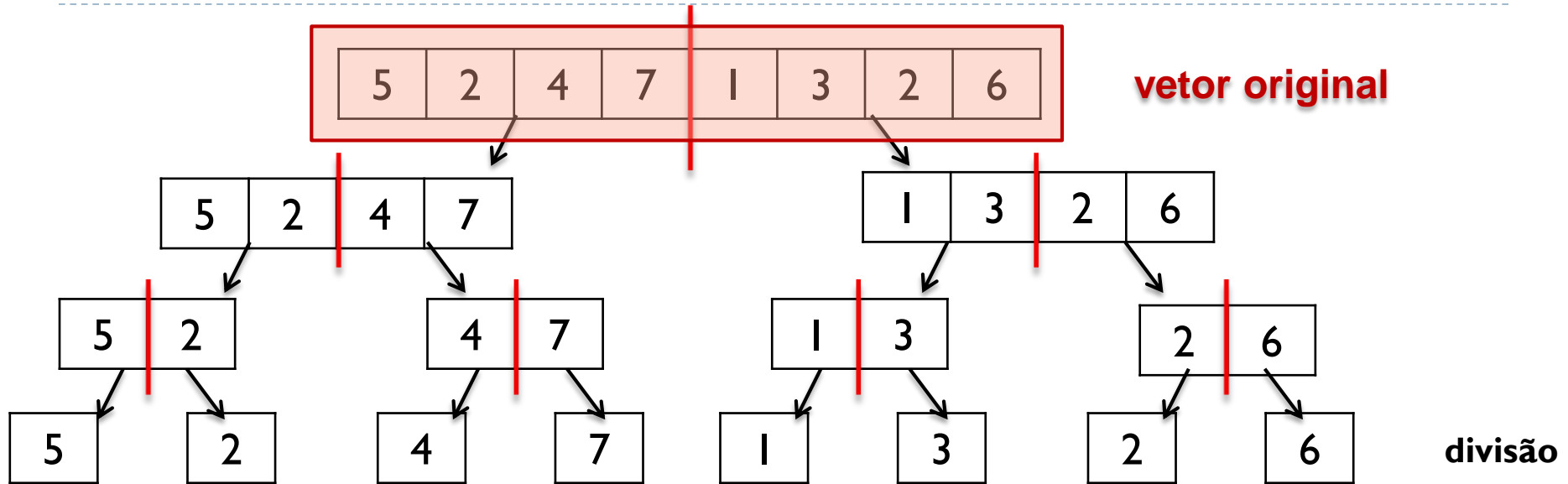
Merge sort : exemplo de funcionamento



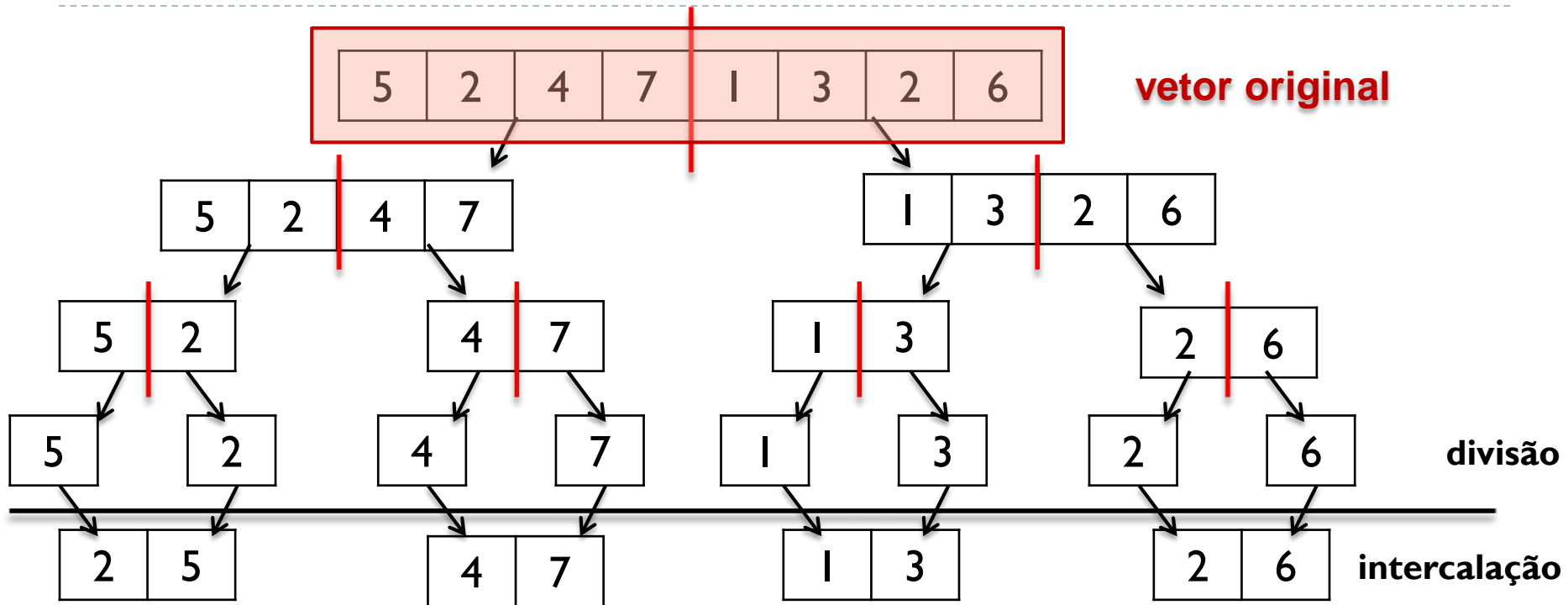
Merge sort : exemplo de funcionamento



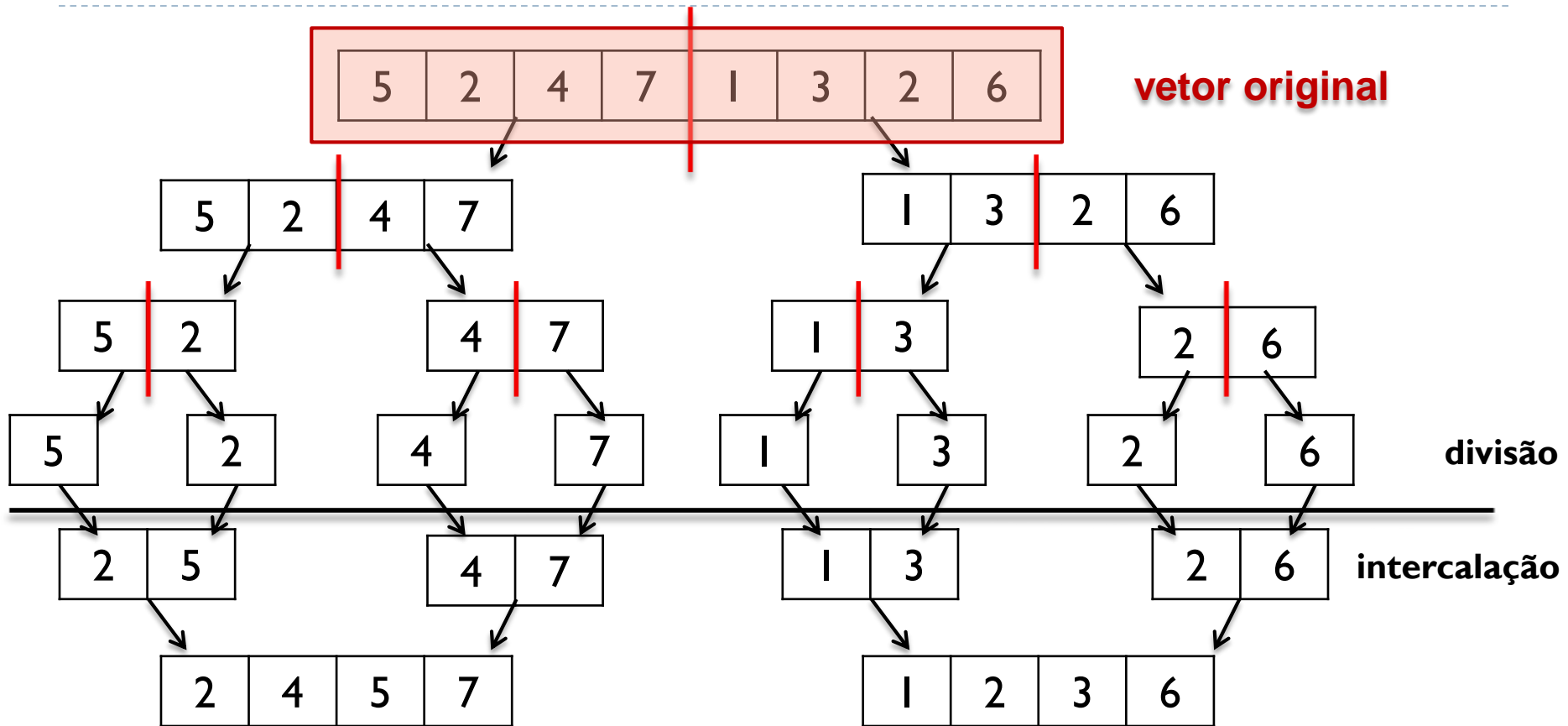
Merge sort : exemplo de funcionamento



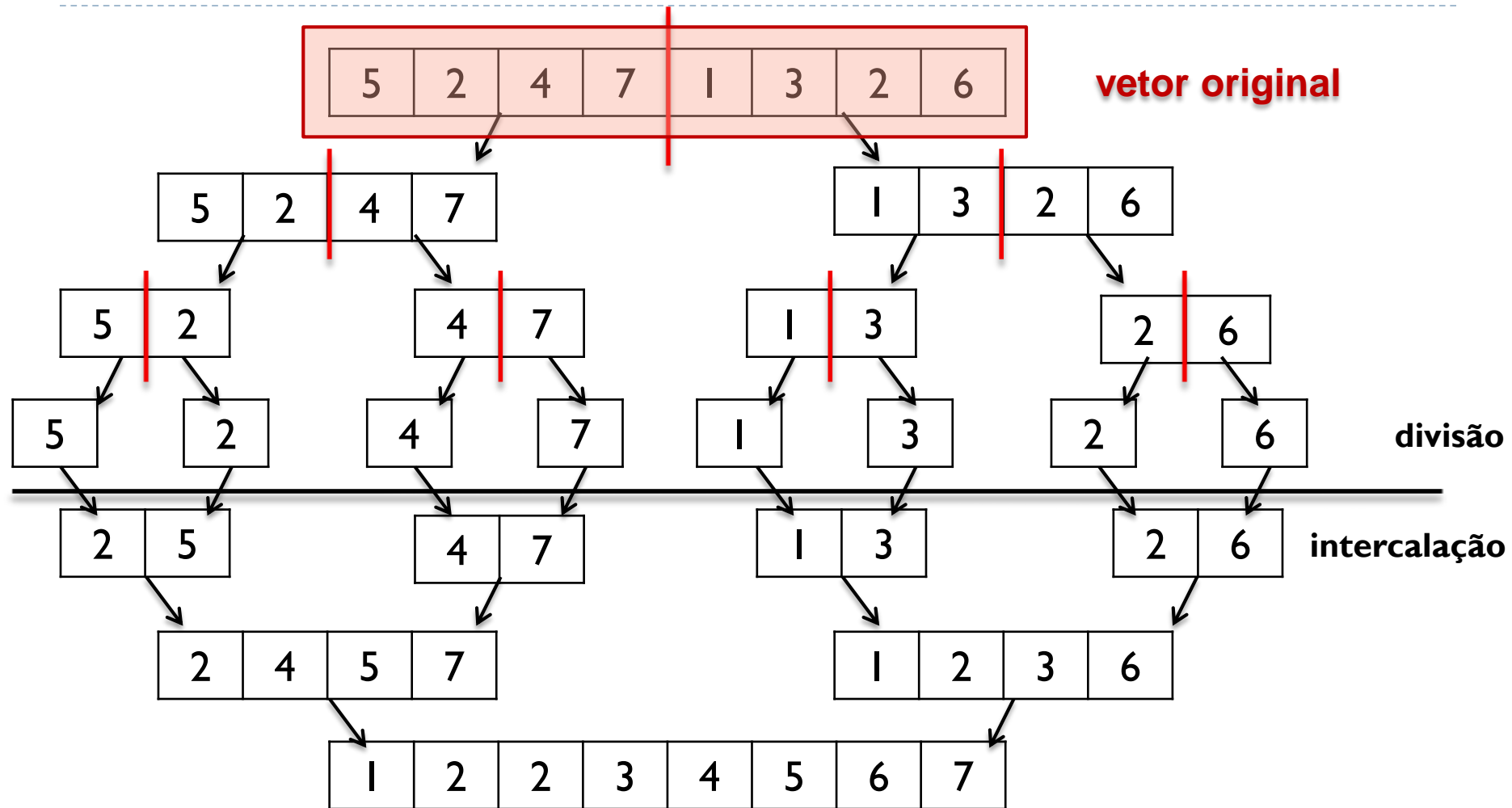
Merge sort : exemplo de funcionamento



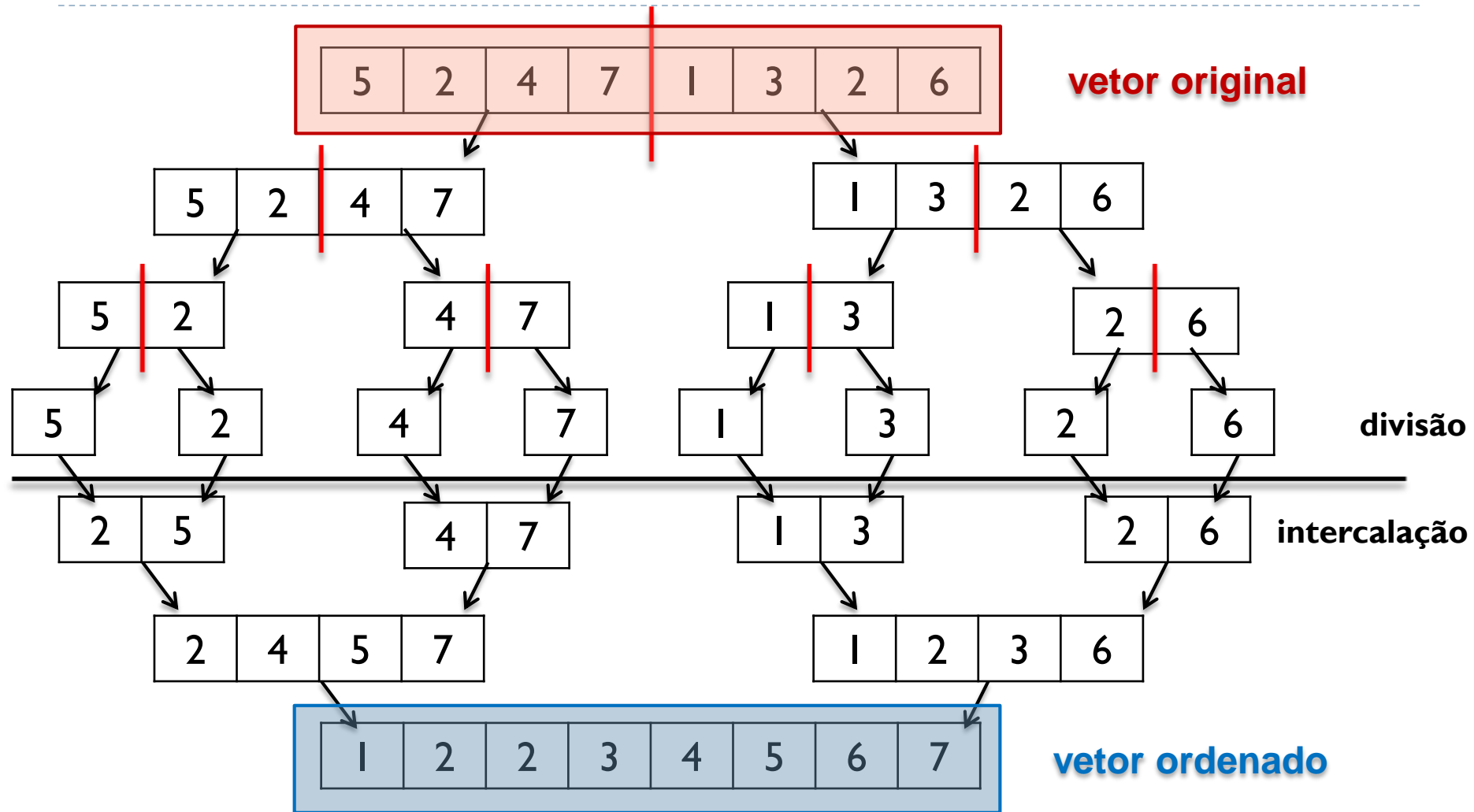
Merge sort : exemplo de funcionamento



Merge sort : exemplo de funcionamento



Merge sort : exemplo de funcionamento



Merge sort : implementação

```
void mergesort (int vetor[], int ini, int fim) {
```

```
    if (inicio < fim) {
```

```
        int meio = floor((inicio+fim)/2);
```

```
        mergesort(vetor, inicio, meio);
```

```
        mergesort(vetor, meio+1, fim);
```

```
        intercala(vetor, inicio, meio, fim);
```

```
    }
```

```
}
```

Merge sort : implementação

```
void mergesort (int vetor[], int ini, int fim) {
```

```
    if (inicio < fim) {
```

```
        int meio = floor((inicio+fim)/2);
```

```
        mergesort(vetor, inicio, meio);
```

```
        mergesort(vetor, meio+1, fim);
```



Chama a função para as 2 partições
(passo recursivo)

```
        intercala(vetor, inicio, meio, fim);
```

```
    }
```

```
}
```

Merge sort : implementação

```
void mergesort (int vetor[], int ini, int fim) {
```

```
    if (inicio < fim) {
```

```
        int meio = floor((inicio+fim)/2);
```

```
        mergesort(vetor, inicio, meio);
```

```
        mergesort(vetor, meio+1, fim);
```

```
        intercala(vetor, inicio, meio, fim);
```

```
    }
```

```
}
```



Chama a função para as 2 partições
(passo recursivo)



Mescla as 2 partições a fim de
garantir a ordenação

Merge sort : implementação

```
void intercala (int *vetor, int ini, int meio, int fim) {
    int p1 = ini, p2 = meio+1, fim1 = 0, fim2 = 0;
    int tamanho = fim - ini + 1;
    int *vaux = (int*) malloc(tamanho*sizeof(int));

    if (vaux != NULL) {
        int i, j, k;
        // Intercala em um vetor aux
        for (i=0; i < tamanho; i++)

            if (! fim1 && ! fim2) {
                if (vetor[p1] < vetor[p2])
                    vaux[i] = vetor[p1++];
                else
                    vaux[i] = vetor[p2++];
                if (p1 > meio) fim1 = 1;
                if (p2 > fim)  fim2 = 1;
            }
            ...
            else {
                if (! fim1)
                    vaux[i] = vetor[p1++];
                else
                    vaux[i] = vetor[p2++];
            }
        // Copia vetor aux para o original
        for (j=0, k=ini; j < tamanho; j++, k++)
            vetor[k] = vaux[j];
    }
    free(vaux);
}
```

Merge sort : análise do algoritmo

- ▶ Qual é a complexidade assintótica do algoritmo?

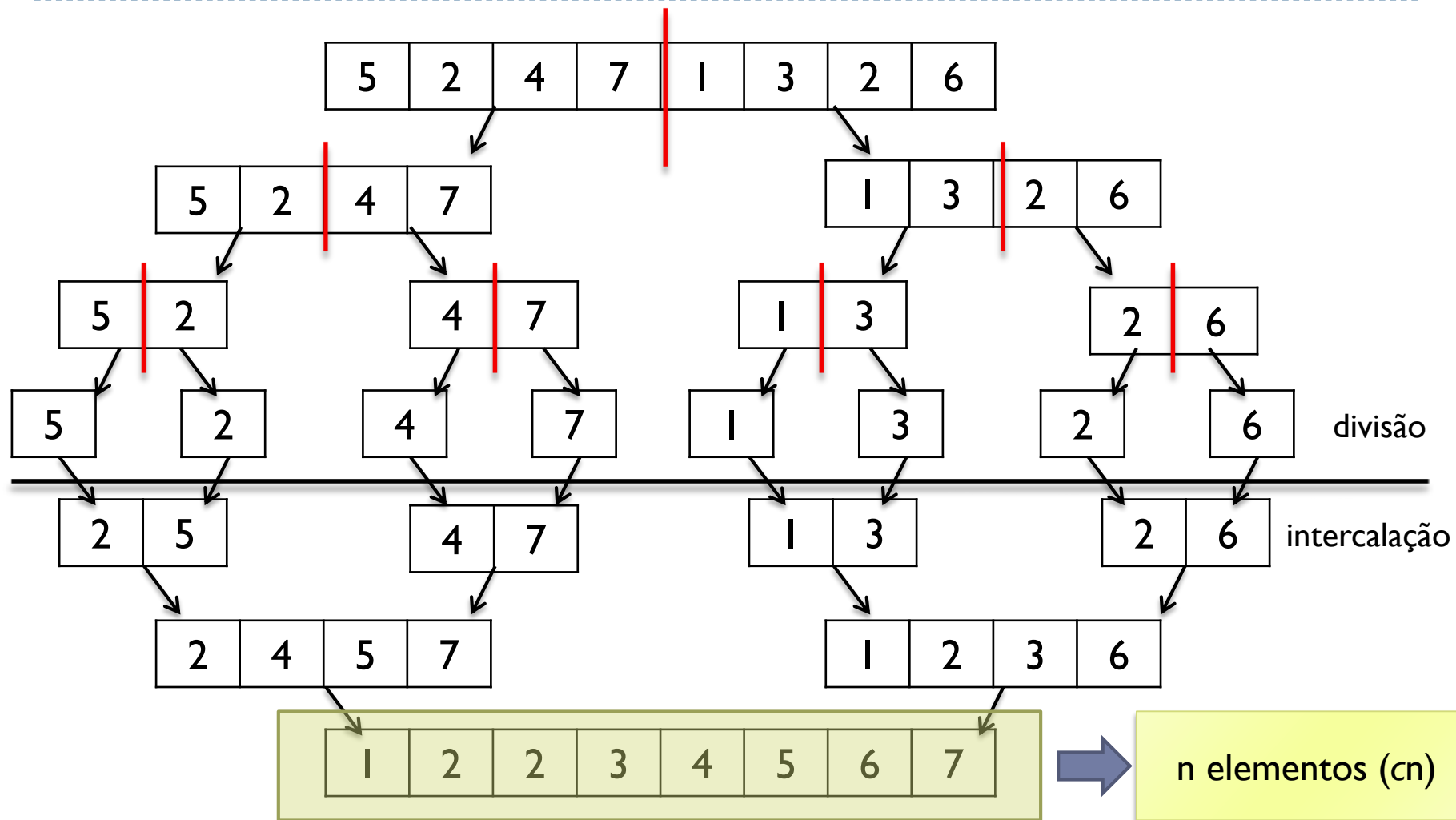
Merge sort : análise do algoritmo

- ▶ Qual é a complexidade assintótica do algoritmo?
- ▶ Resposta depende da análise do tempo gasto:
 - ▶ Particionamento do arquivo (**divisão**)

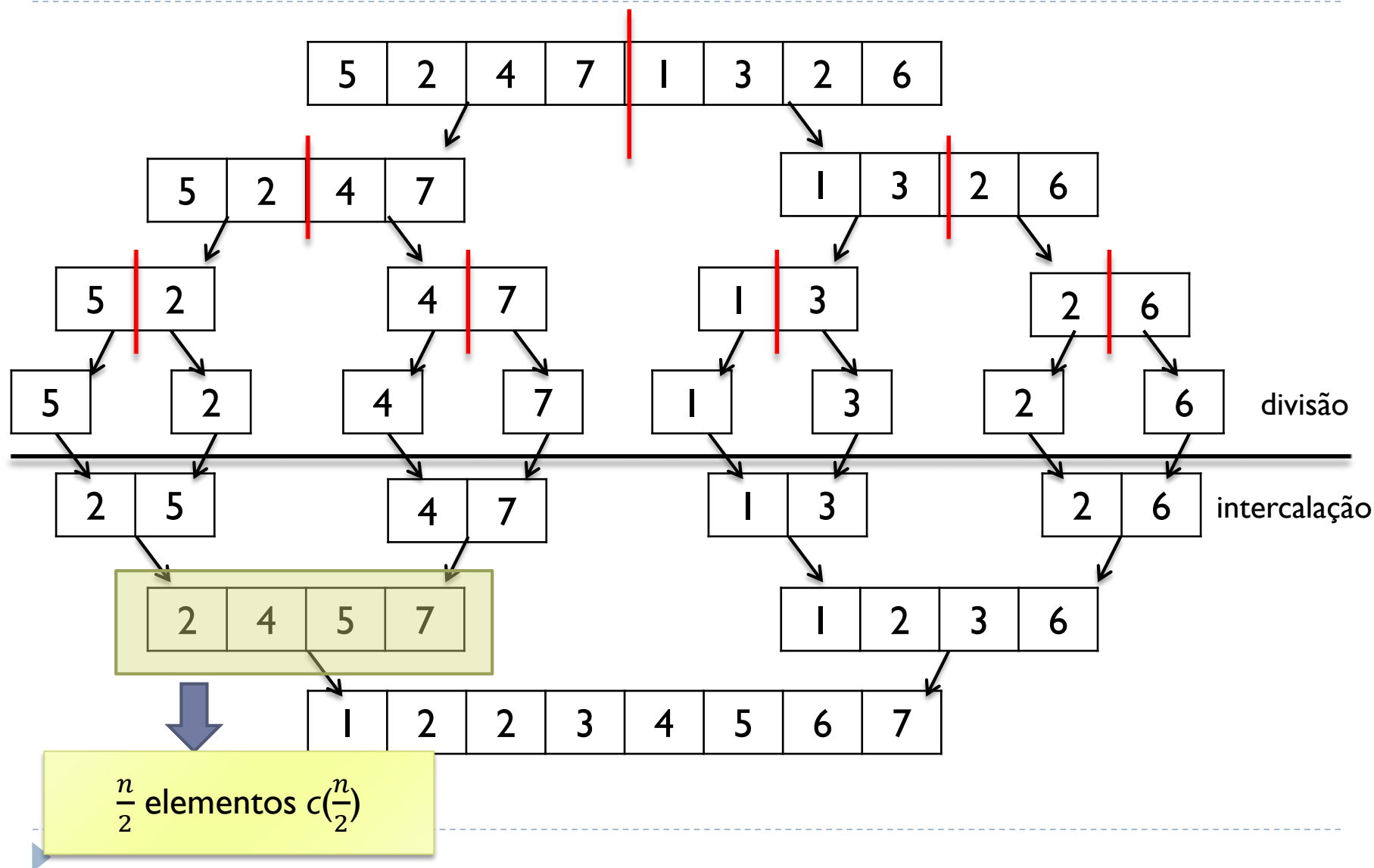
Merge sort : análise do algoritmo

- ▶ Qual é a complexidade assintótica do algoritmo?
- ▶ Resposta depende da análise do tempo gasto:
 - ▶ Particionamento do arquivo (**divisão**)
 - ▶ **Intercalação** das partições

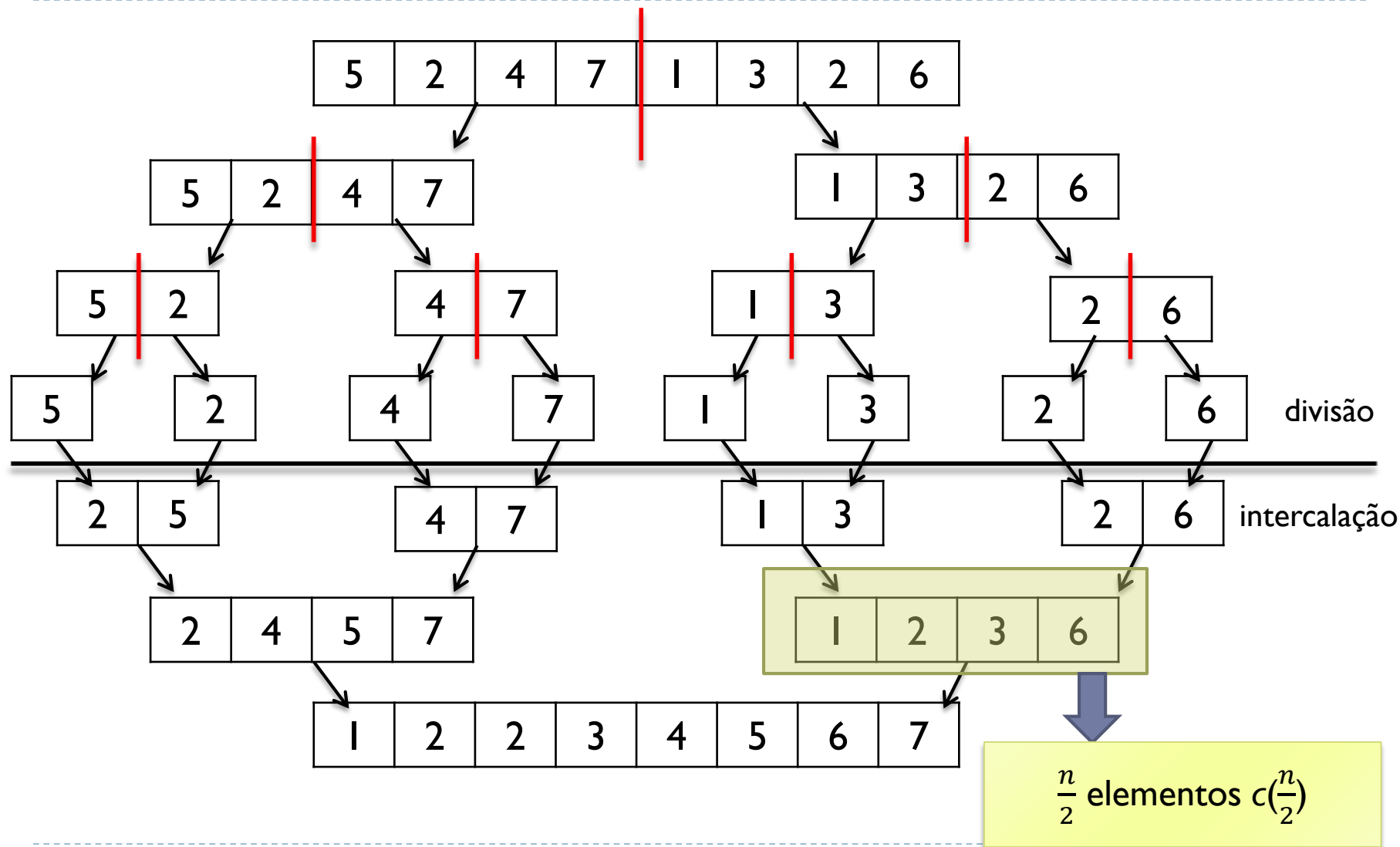
Merge sort : análise da intercalação



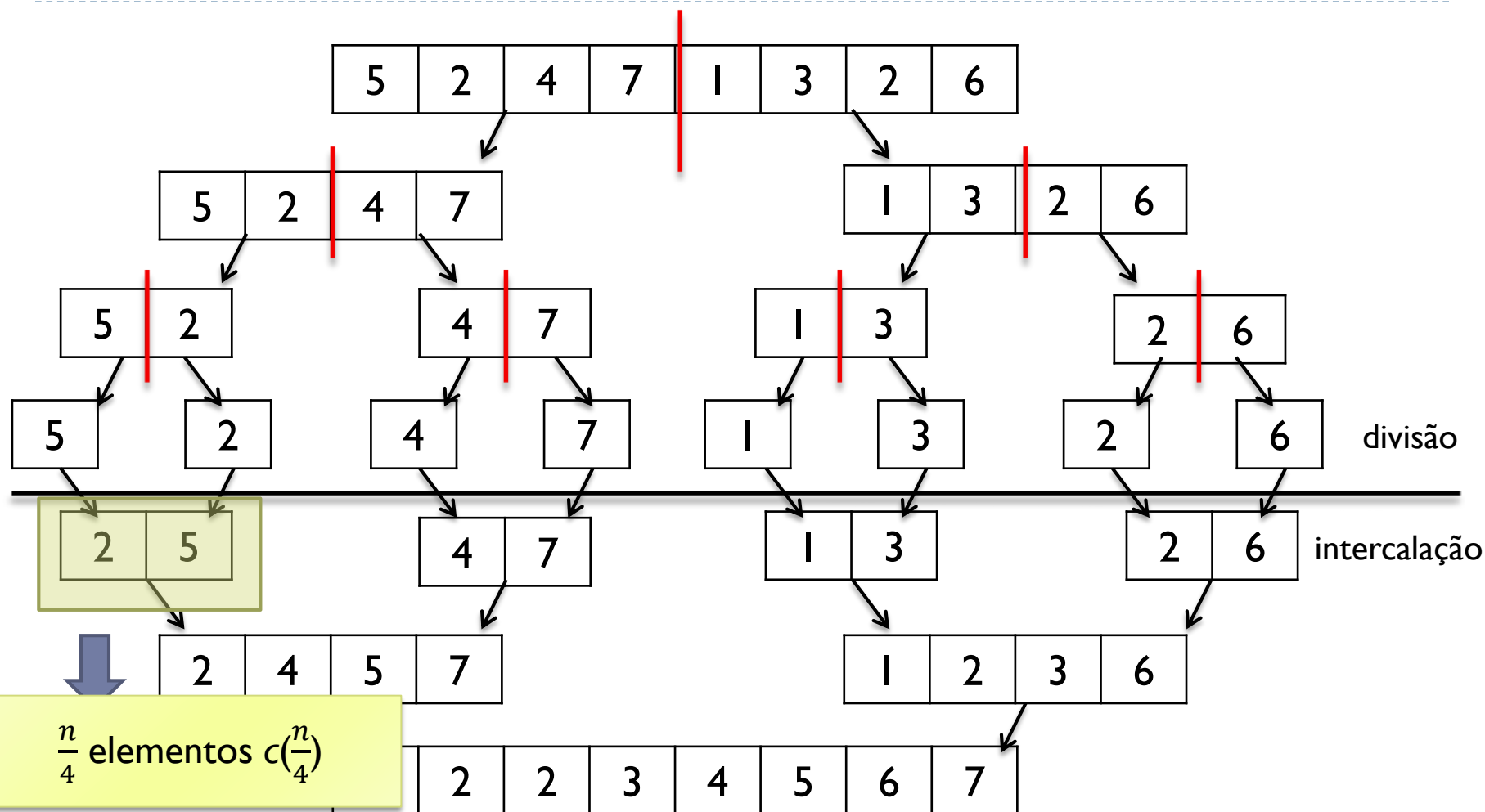
Merge sort : análise da intercalação



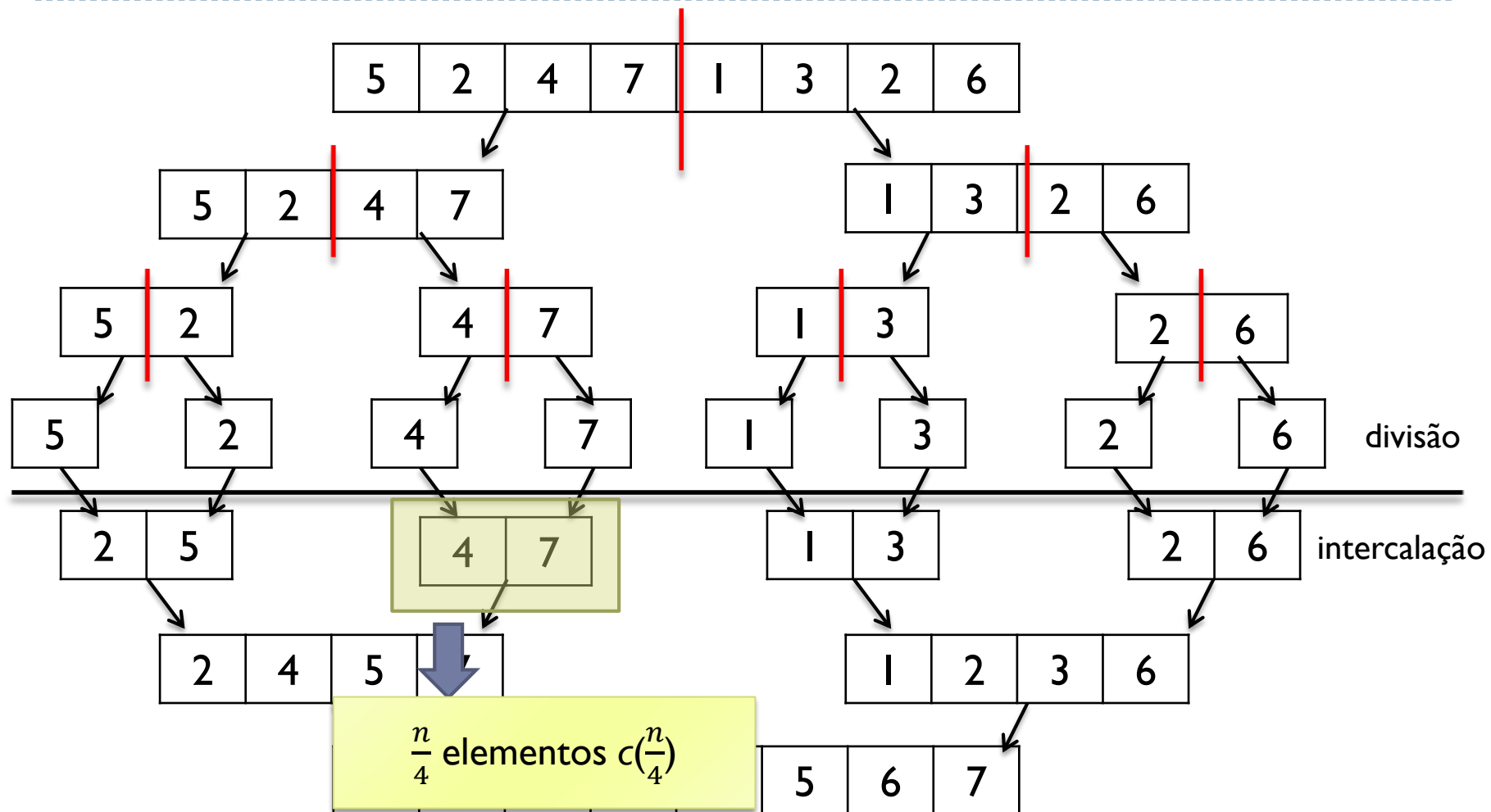
Merge sort : análise da intercalação



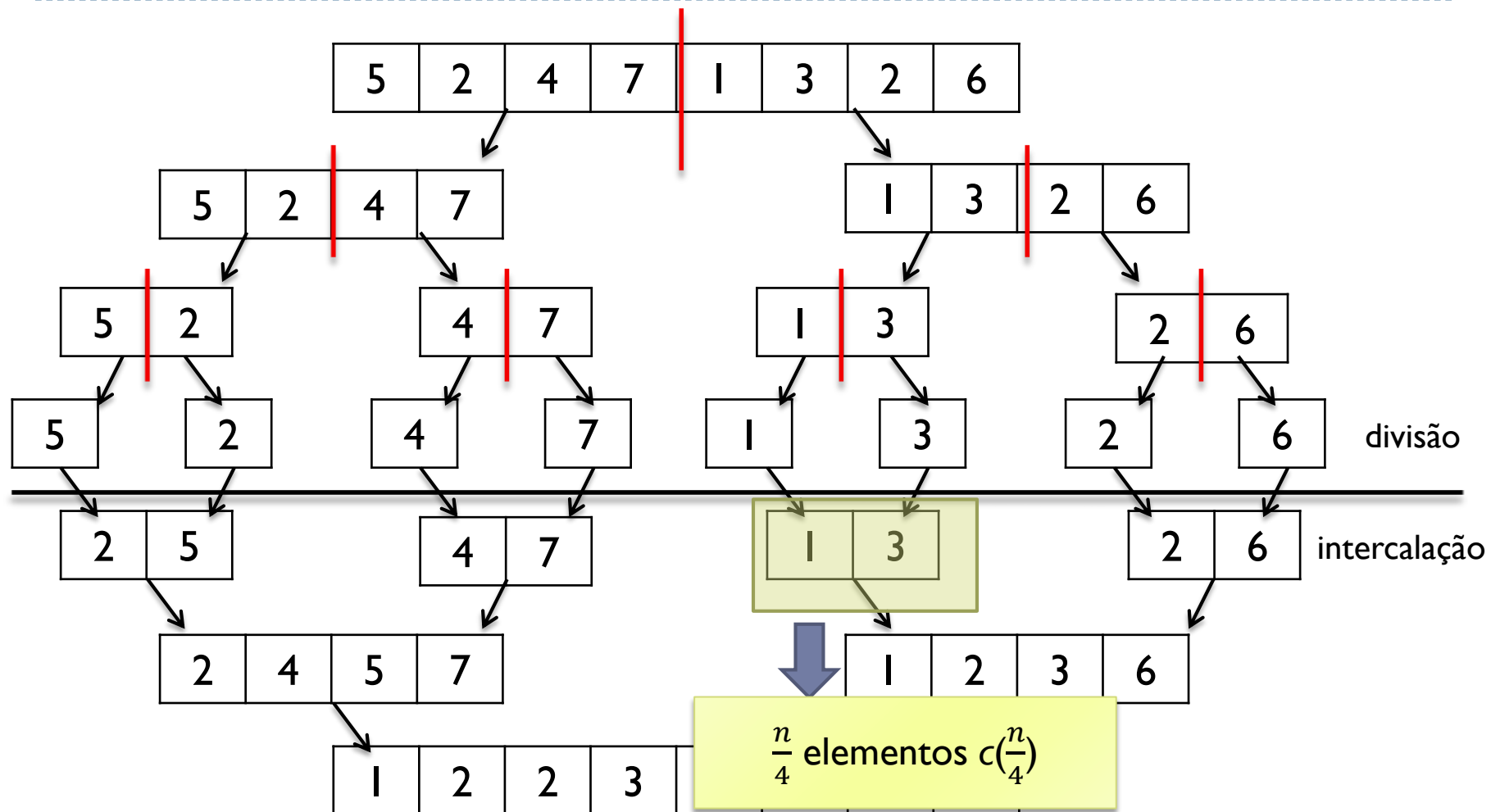
Merge sort : análise da intercalação



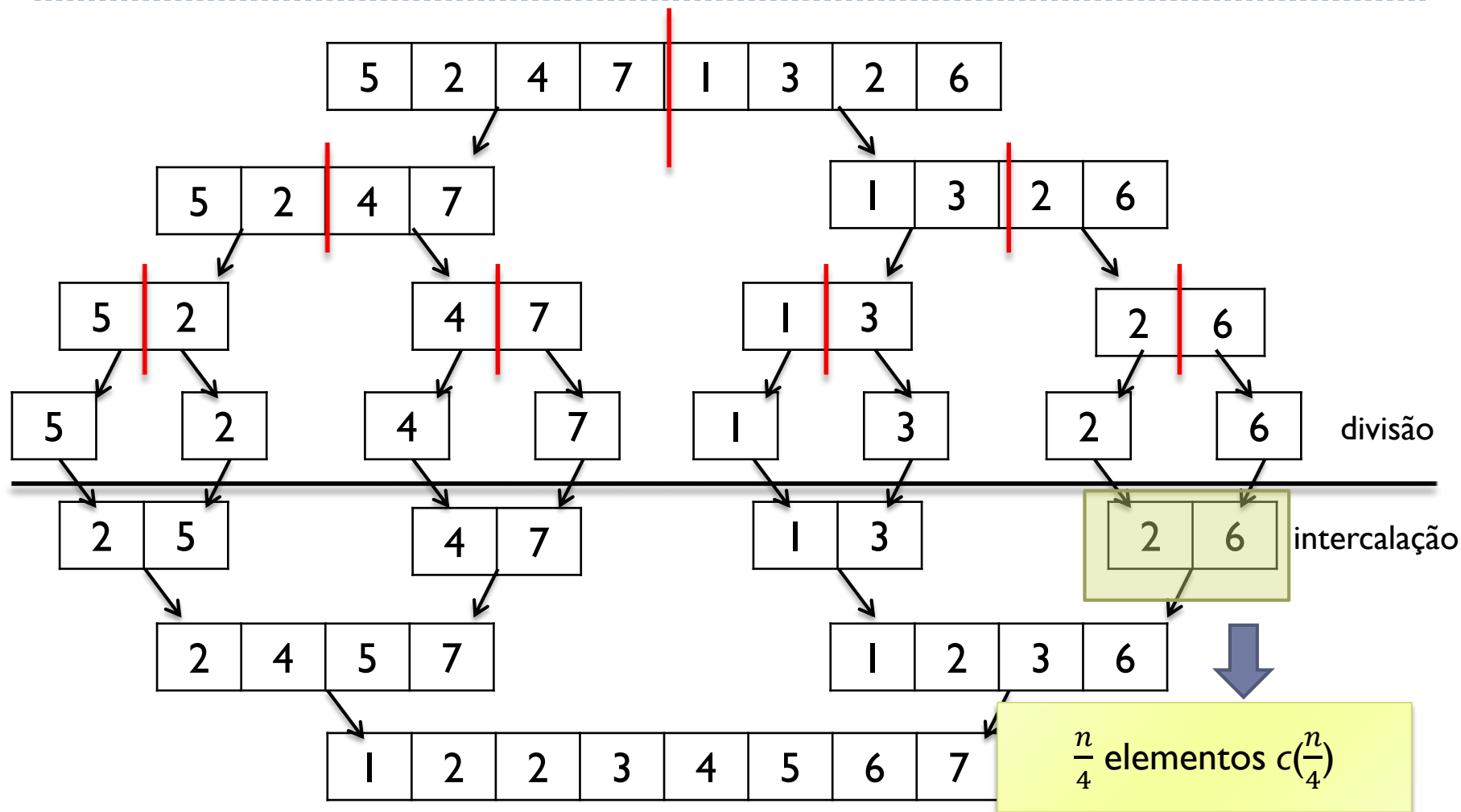
Merge sort : análise da intercalação



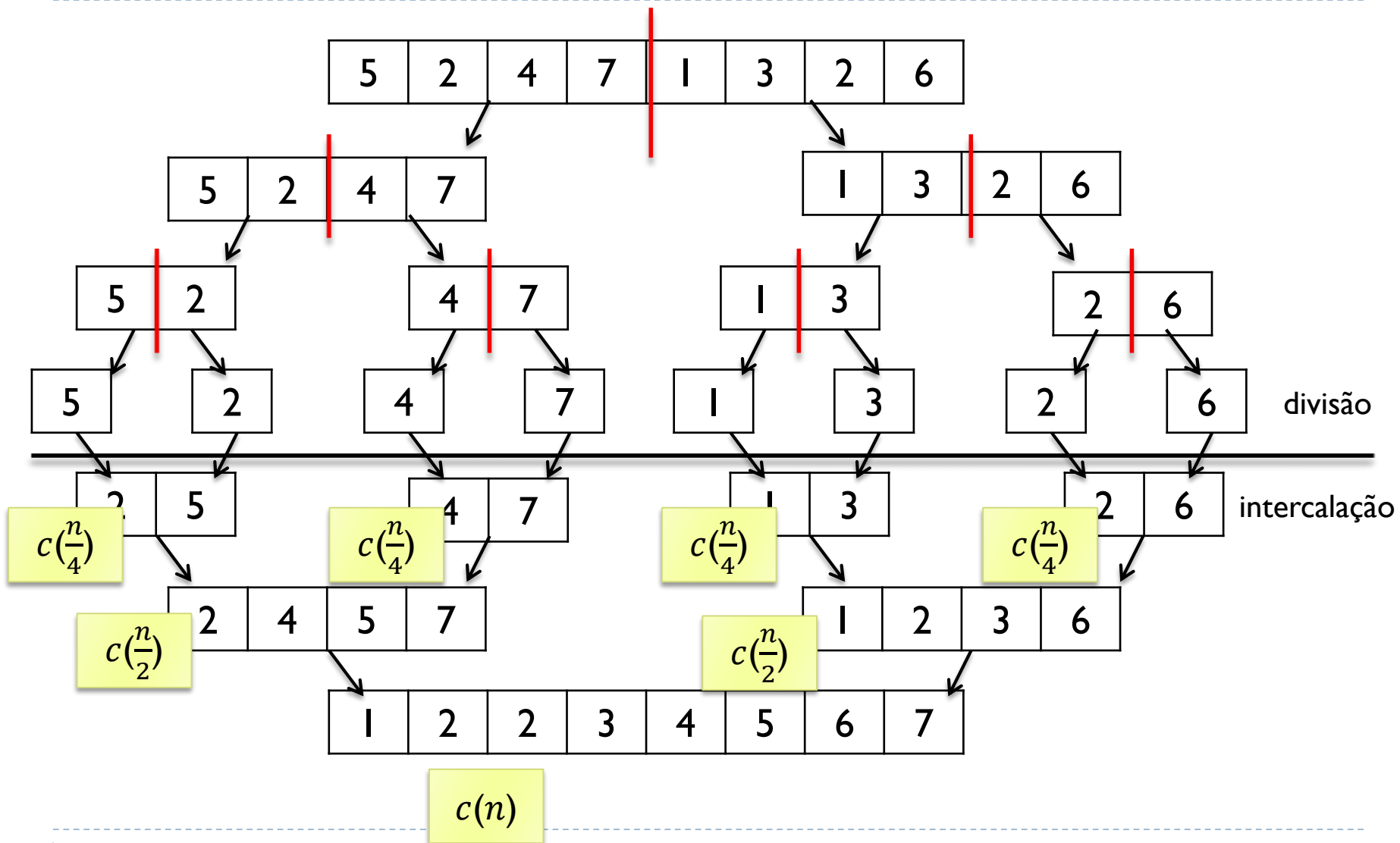
Merge sort : análise da intercalação



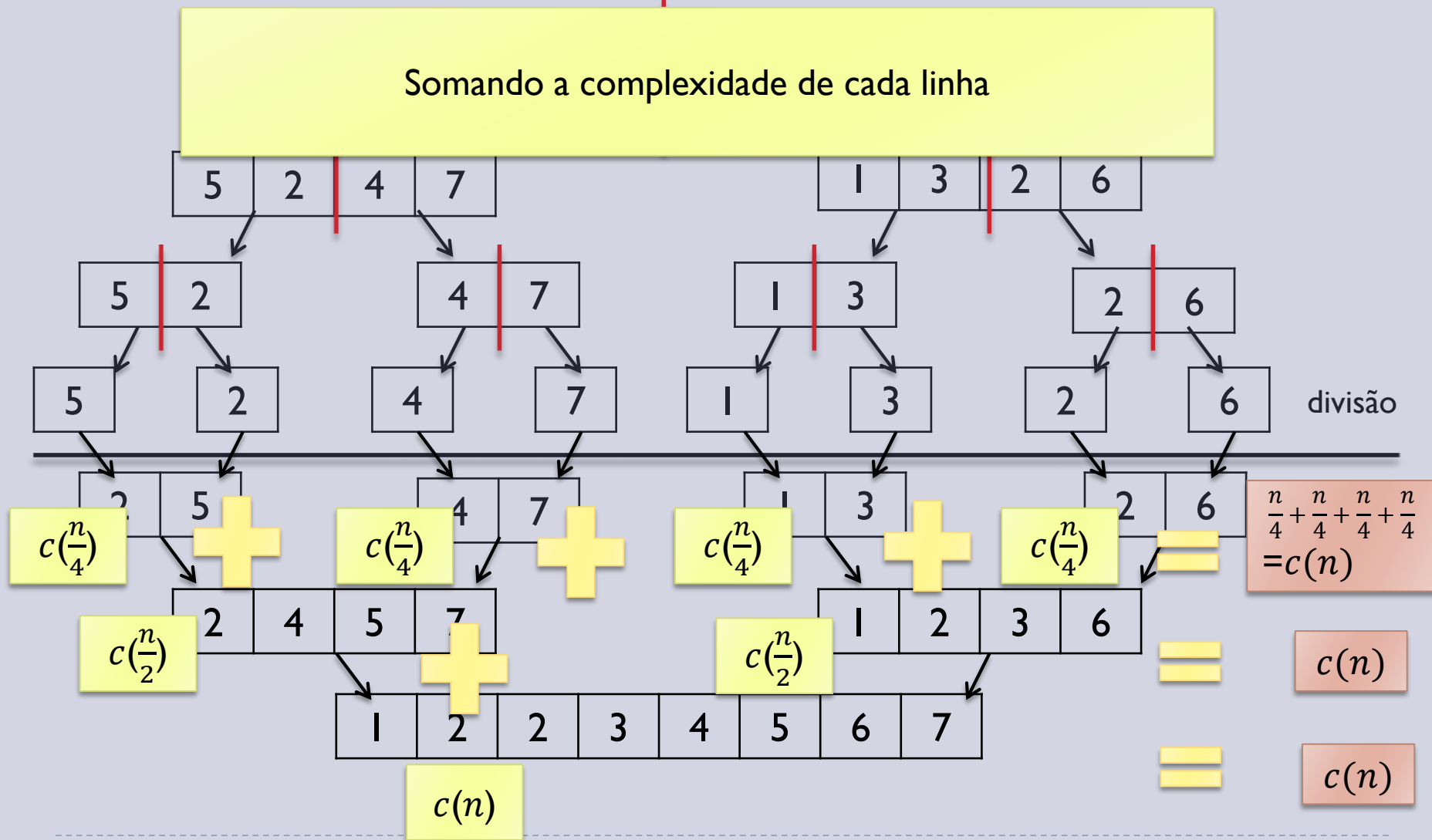
Merge sort : análise da intercalação



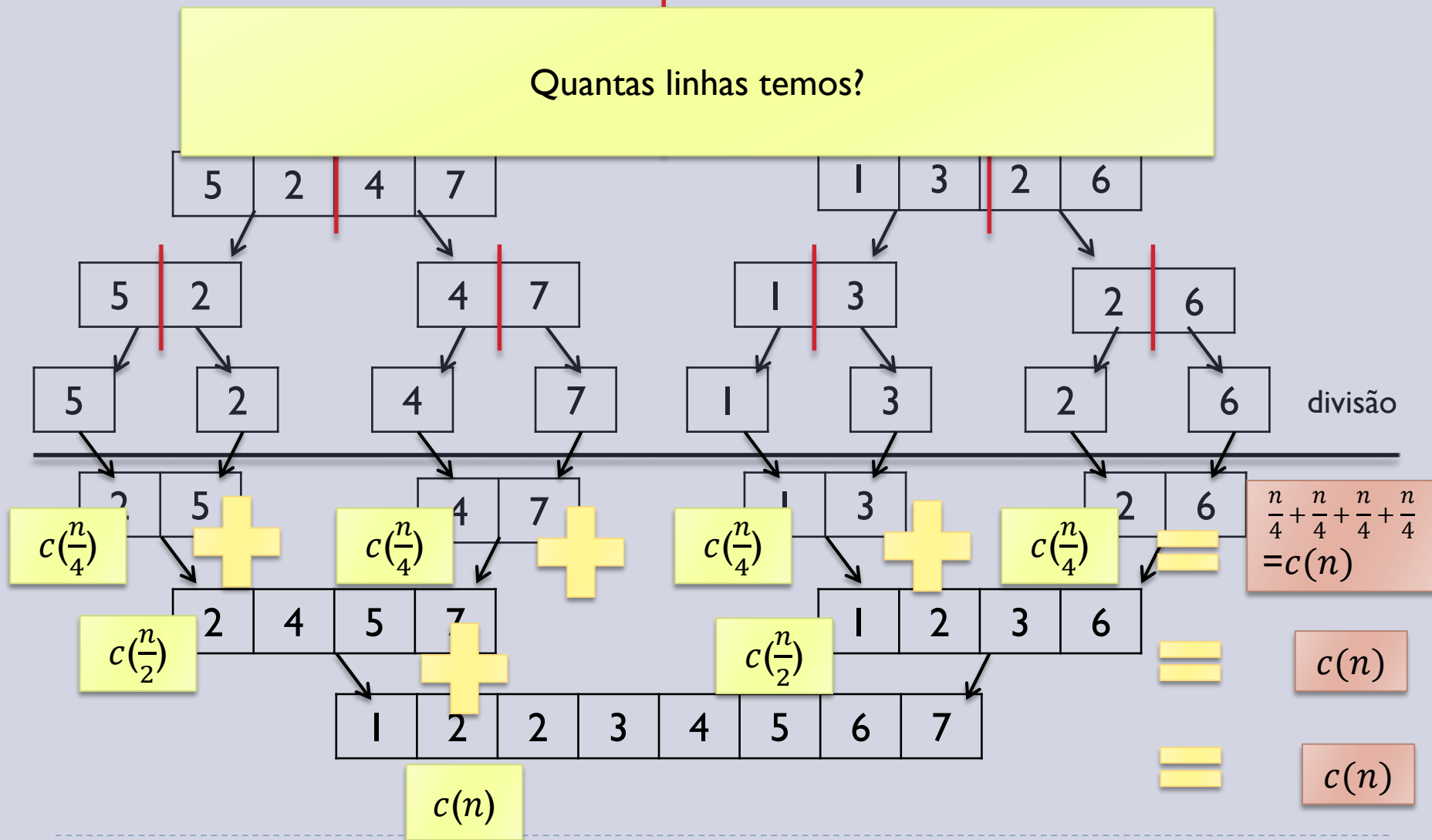
Merge sort : análise da intercalação



Merge sort : análise da intercalação

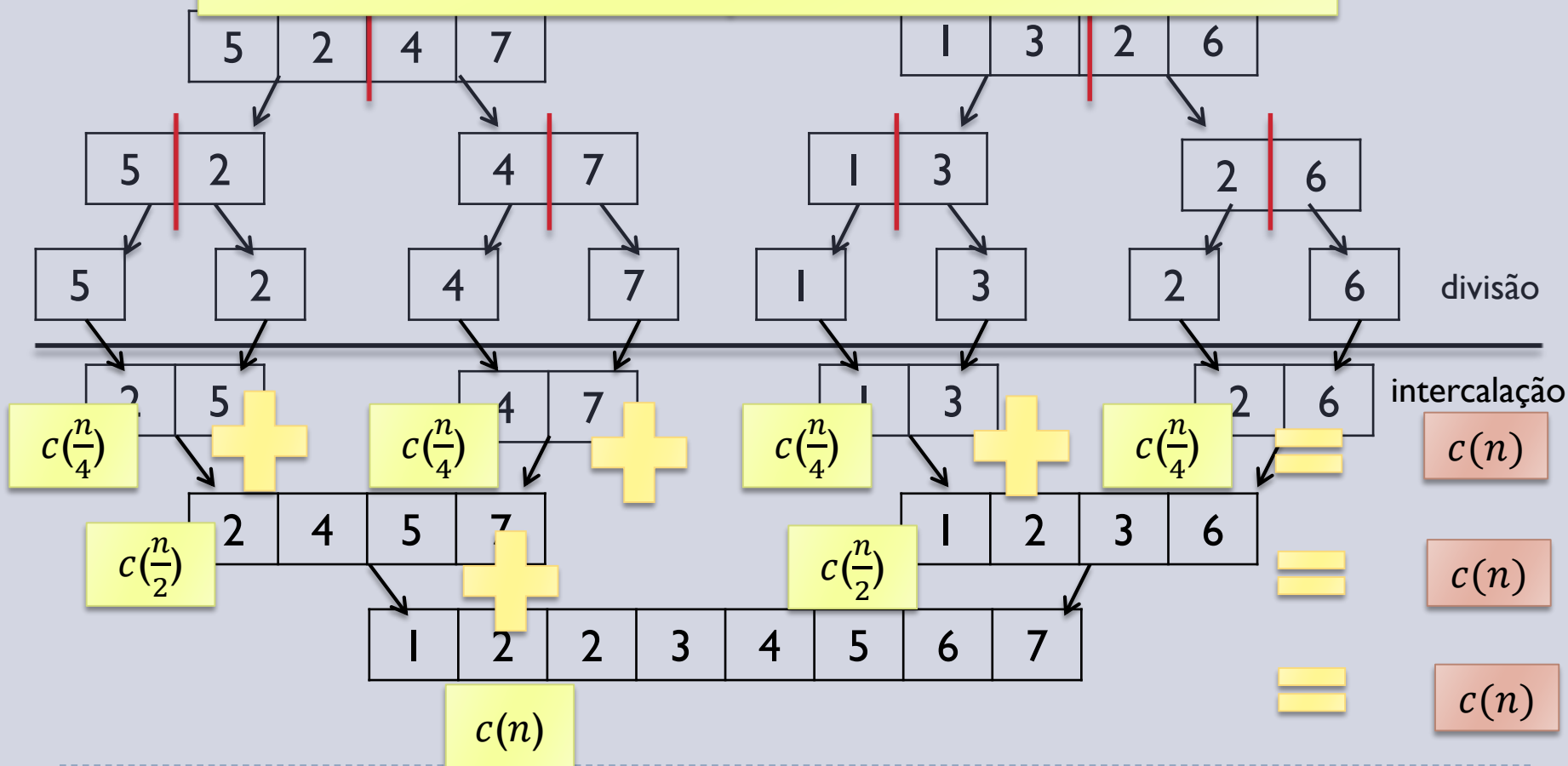


Merge sort : análise da divisão



Merge sort : análise da divisão

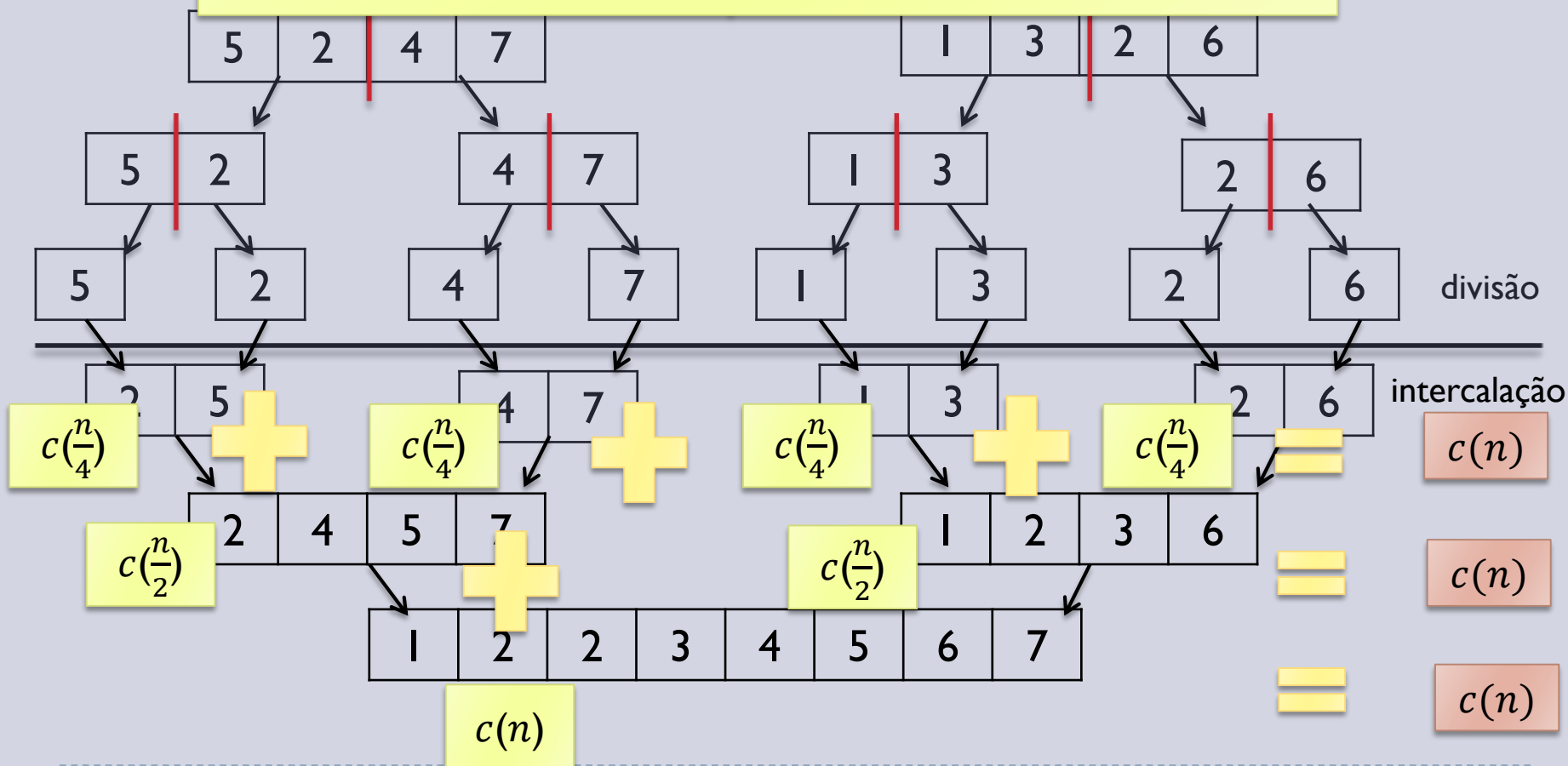
Quantas vezes podemos dividir ao meio um vetor de tamanho n ?



Merge sort : análise da divisão

Quantas vezes podemos dividir ao meio um vetor de tamanho n ?

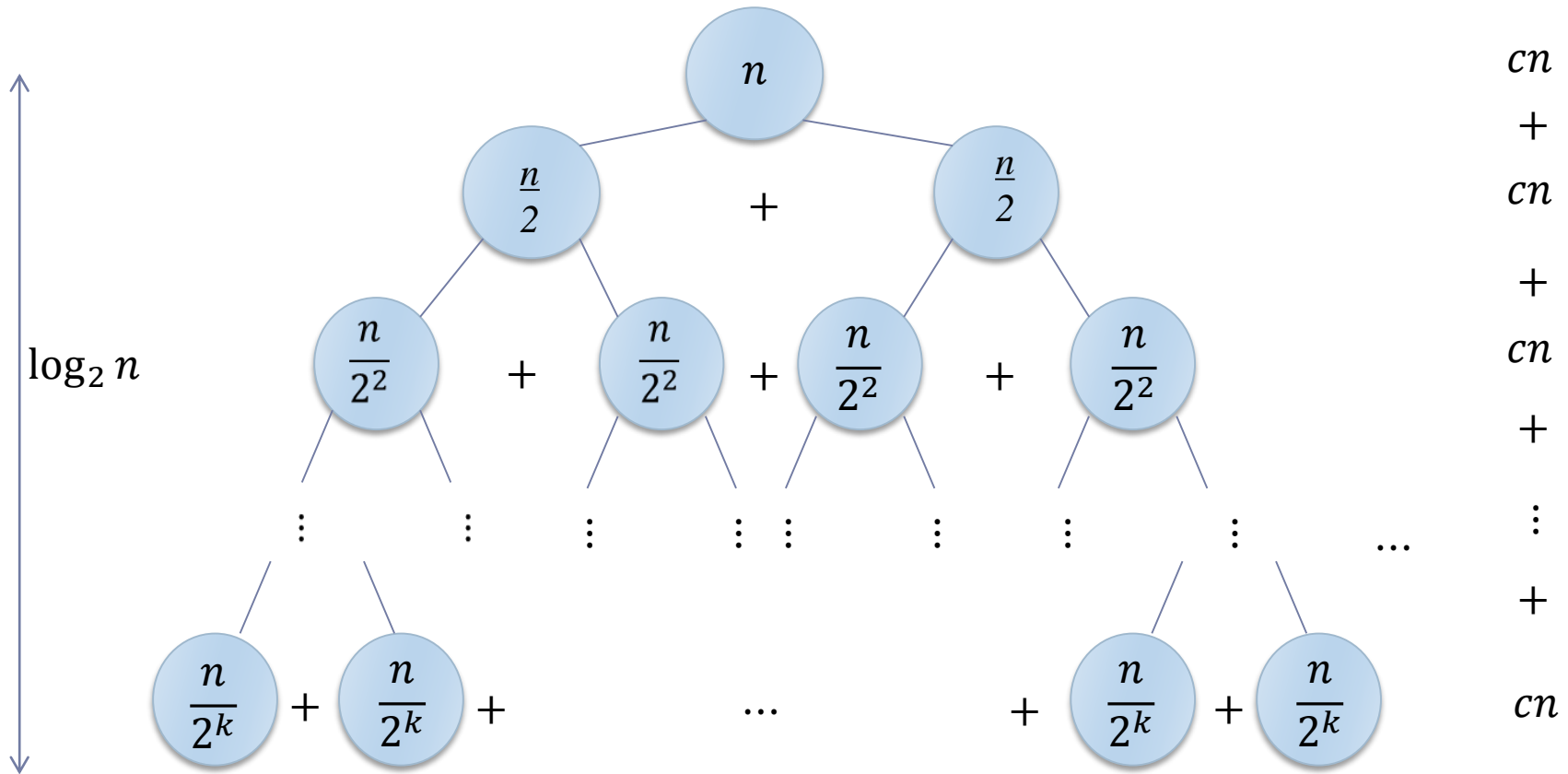
$\log_2 n$



Merge sort : análise do algoritmo

- ▶ Complexidade assintótica:
 - ▶ Divisão é feita em $O(\log n)$
 - ▶ Cada nível de intercalação é feita em $O(n)$
 - ▶ Custo total = $O(n) \times O(\log n) = O(n \log n)$

Merge sort : análise do algoritmo



$$cn \log_2 n \approx O(n \log n)$$

Resumo (<http://bigocheatsheet.com/>)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$

Obs: $O(n)$ é obtida na versão modificada do *bubble sort* que verifica se não houve trocas na iteração, aplicada sobre um vetor já ordenado

Bucket sort

- ▶ Algoritmo simples para ordenação de números inteiros
- ▶ **Ideia:**
 - ▶ Cada elemento é representado por uma posição em um arranjo (**vetor de recipientes**)
 - ▶ As repetições de um mesmo número são acumuladas em um recipiente
 - ▶ Ao final, o conteúdo de cada recipiente é lido de modo sequencial e seu índice é usado para preencher o vetor de saída (**vetor ordenado**)
 - ▶ Valor indica a qtde de vezes que o índice será usado

Bucket sort : implementação

```
void bucketsort (int *vetor, int n, int w){
    int *vaux = (int *) malloc((w)*sizeof(int));
    if (vaux) {
        // Preenchimento dos recipientes
        for(int i =0; i < n; i++)
            vaux[vetor[i]]++;

        // Leitura dos recipientes em ordem
        int i =0;
        for(int j = 0; j < w; j++)
            while(vaux[j] > 0) {
                vaux[j] = vaux[j]-1;
                vetor[i] = j;
                i=i+1;
            }
    }
}
```

w é o **tamanho máximo**
dos números

Bucket sort : complexidades

- ▶ **Complexidade de tempo:**

- ▶ Cada número é avaliado **uma única vez**
- ▶ Custo **$O(n)$**

Bucket sort : complexidades

▶ **Complexidade de tempo:**

- ▶ Cada número é avaliado **uma única vez**
- ▶ Custo **$O(n)$**

▶ **Complexidade de espaço:**

- ▶ Viável apenas para inteiros ou números com poucas casas decimais
- ▶ Cresce com a faixa de valores considerada
- ▶ **Ex:** Se w é a **qtde. máxima de algarismos** dos números, então a complexidade é **$O(10^w)$**

Bucket sort : complexidade de espaço

- ▶ **Ex:** ordenação de 40 milhões de transações financeiras
 - ▶ Se 1% das transações forem superiores a 1 milhão de reais, então é viável usar 2 algoritmos de ordenação:
 - ▶ Transações de até 1 milhão - ***bucket sort*** (4 segundos)
 - ▶ Transações superiores - ***bubble sort*** (2,5 minutos)

Bucket sort : complexidade de espaço

- ▶ **Ex:** ordenação de 40 milhões de transações financeiras
 - ▶ Se 1% das transações forem superiores a 1 milhão de reais, então é viável usar 2 algoritmos de ordenação:
 - ▶ Transações de até 1 milhão - **bucket sort** (4 segundos)
 - ▶ Transações superiores - **bubble sort** (2,5 minutos)
- ▶ **Balanceamento de complexidades:**
 - ▶ **Bucket sort:** complexidade de tempo baixa e complexidade de espaço alta
 - ▶ **Bubble sort:** complexidade de tempo alta e complexidade de espaço baixa

Exercícios

1. Comparar os métodos de ordenação (exceto o *bucket sort*) em termos de tempo de execução, de número de comparações e de número de trocas.
2. Faça a análise empírica dos métodos de ordenação (exceto o *bucket sort*), utilizando arranjos de 100, 1000 e 10000 números inteiros entre 0 e 500. Considere 3 configurações:
 - ▶ Arranjo com elementos em ordem crescente
 - ▶ Arranjo com elementos em ordem decrescente
 - ▶ Arranjo com elementos em ordem aleatória
3. Faça uma implementação recursiva dos métodos de ordenação simples (bolha, seleção e inserção). Analise a complexidade dessas implementações.

Exercícios

5. Modifique o algoritmo do *quick sort* de modo a adotar a ordenação por inserção quando uma partição tiver tamanho abaixo de s . Por fim, determine através de uma análise empírica qual valor de s deve ser adotado para alcançar a melhor eficiência.
6. Os algoritmos apresentados realizam uma ordenação destrutiva, na qual o arranjo original é perdido, sendo substituído pelo arranjo ordenado. Uma boa alternativa é a ordenação indireta, realizada através de uma tabela auxiliar que indica a posição do elemento no arranjo original. Implemente um programa que realize esse tipo de ordenação usando o método *select sort*.
7. Pesquise e implemente o método *shell sort*. Depois, faça uma análise comparativa com a ordenação por inserção.

Exercício 3: *bubble sort* recursivo

```
void bubble_rec (int vetor[], int n){  
    int i, aux, troca = 0;  
  
    for (i=0; i < n-1; i++)  
  
        if (vetor[i] > vetor[i+1]) {  
            aux = vetor[i];  
            vetor[i] = vetor[i+1];  
            vetor[i+1] = aux;  
            troca = 1;  
        }  
  
    if (troca != 0)  
        bubble_rec (n-1, vetor);  
}
```

Exercício 3: *bubble sort* recursivo

```
void bubble_rec (int vetor[], int n){  
    int i, aux, troca = 0;
```

```
    for (i=0; i < n-1; i++)
```

```
    {  
        if (vetor[i] > vetor[i+1]) {  
            aux = vetor[i];  
            vetor[i] = vetor[i+1];  
            vetor[i+1] = aux;  
            troca = 1;  
        }  
    }
```

Iteração mais interna
(**comparação dos pares de elementos**)

```
    if (troca != 0)  
        bubble_rec (n-1, vetor);  
}
```


Exercício 3: *bubble sort* recursivo

```
void bubble_rec (int vetor[], int n){  
    int i, aux, troca = 0;
```

```
    for (i=0; i < n-1; i++)
```

```
    {  
        if (vetor[i] > vetor[i+1]) {  
            aux = vetor[i];  
            vetor[i] = vetor[i+1];  
            vetor[i+1] = aux;  
            troca = 1;  
        }  
    }
```

Iteração mais interna
(comparação dos pares de elementos)

```
    if (troca != 0)  
        bubble_rec (n-1, vetor);
```

Passo recursivo
(existência de trocas na iteração)

```
}
```

Bibliografia

- ▶ Slides adaptados do material do Prof. Dr. Bruno Travençolo, do Prof. Autran Macêdo e da Profa. Dra. Denise Guliato.
- ▶ BACKES, A. Linguagem C Descomplicada: portal de vídeo-aulas para estudo de programação. Disponível em: <https://programacaodescomplicada.wordpress.com/indice/estrutura-de-dados/>
- ▶ CORMEN, T.H. et al. Algoritmos: Teoria e Prática, Campus, 2002
- ▶ ZIVIANI, N. Projeto de algoritmos: com implementações em Pascal e C (2ª ed.), Thomson, 2004

Bibliografia

- ▶ MORAES, C.R. Estruturas de Dados e Algoritmos: uma abordagem didática (2ª ed.), Futura, 2003
- ▶ FEOFIOFF, P. **Quicksort**. Disponível em:
<http://www.ime.usp.br/~pf/algoritmos/aula/quick.html>
- ▶ SHEWCHUCK, J. **Data Structures**. Disponível em:
<http://www.cs.berkeley.edu/~jrs/61bs09/>